

Courant Institute of
Mathematical Sciences

AEC Computing and Applied Mathematics Center

A Preliminary Report on CIMS PL/I

Paul Abrahams

AEC Research and Development Report

Mathematics and Computers
August 1973



New York University

COO-3077-31
Abrahams

UNCLASSIFIED

AEC Computing and Applied Mathematics Center
Courant Institute of Mathematical Sciences
New York University

Mathematics and Computers

COO-3077-31

A PRELIMINARY REPORT ON CIMS PL/I

Paul Abrahams

Contract No. AT(11-1)-3077

UNCLASSIFIED

<u>Table of Contents</u>	<u>Page</u>
1. Introduction.....	1
2. Implementation Characteristics.....	2
2.1 Language accepted by CIMS PL/I.....	2
2.2 Features Not Yet Implemented.....	4
2.3 Operating Characteristics.....	6
3. Bootstrapping Method.....	7
4. Structure of the Compiler.....	11
5. Implementation Techniques.....	14
5.1 Parsing Method.....	14
5.2 Data Structure Design.....	17
5.2.1 Choice of Representation Method.....	17
5.2.2 Examination of Trade-Offs for Specific Structures.....	19
5.3 Run-Time Storage Allocation.....	22
6. Conclusions.....	24
6.1 Sources of Program Complexity.....	24
6.2 Operating Systems Support.....	27
6.3 Virtual Memory.....	29
6.4 Reliability.....	30
6.5 Project Organization.....	31
6.6 PL/I as a Systems Programming Language.....	32
6.7 Design Flaws.....	33
References.....	36
Appendix A - User's Manual for CIMS PL/I.....	37
Appendix B - Useful Coding Tricks.....	64

1. Introduction

CIMS PL/I is an implementation of a PL/I subset on the CDC 6600 computer. The subset is oriented towards compiler writing, and is sufficiently large so that the compiler itself could be conveniently written in it. The objective of this report is to set forth some of the more interesting aspects of the compiler and some of the conclusions that I drew from creating it.

I had several initial objectives when I decided upon this project. One of them was to create a compiler for a useful programming language. That goal has been achieved, though it is now less significant since CDC has also produced a PL/I compiler. A second objective was to learn about compiling techniques. Compilation of PL/I is sufficiently difficult so that if one can compile PL/I one can probably compile anything. A third objective was to gain more understanding of the characteristics of complex programs. A fourth objective was to stimulate thinking about programming language design.

CIMS PL/I has been operational since October 1972 and consumed about three and a half calendar years and one and a half man years of work. Production of the original version of the compiler was essentially a one-person effort. Since the original version appeared, a number of extensions and improvements have been made, some of them by others. The compiler itself is written in PL/I, and was bootstrapped using the IBM 360 and the IBM F-compiler for PL/I.

There are two appendices to this report. Appendix A is the current user's guide to CIMS PL/I. Appendix B describes a collection of coding tricks that I found useful and that may be useful to others.

2. Implementation Characteristics

Two aspects of the implementation are worth describing: the language accepted by the compiler and the operating characteristics of the compiler. The three following subsections discuss the features included in CIMS PL/I, the features not yet implemented, and the operating characteristics.

2.1 Language Accepted by CIMS PL/I

In this section I shall only summarize the available language features, as a complete description is given in Appendix A. In implementing these features, there are many possible language definitions to which one can turn. I have used as my guide the definition of PL/I being developed as a proposed standard by committee X3J1 of the American National Standards Institute jointly with committee TC10 of the European Computer Manufacturers Association. While the proposed standard has not yet been issued, drafts of it are available [5]. These drafts do not, however, include final decisions on all language questions.

The datatypes that have been implemented so far are FIXED, FLOAT, CHARACTER, BIT, POINTER, AREA, LABEL, FILE, and ENTRY. Labels, files, and entries are presently limited to constants. The implementation of FLOAT is quite weak, since only arithmetic operations are provided for it; in particular, there is as yet no floating point input or output. The only way to create or access floating point constants is through conversion from FIXED.

Various other attributes are available. Arrays and structures have been implemented, and these may be nested within each other. The storage classes other than CONTROLLED are available, as well as the INITIAL attribute (with some restrictions) and the LIKE attribute. Most of the statement types have been implemented. In general, though, the power of a PL/I subset is not accurately reflected by the range of statements that it recognizes.

Transput (usually called input-output) is rather primitive at present. The only transput operations are writing a character

string to a dataset or reading a character string from a dataset. Datasets can be rewound by opening and closing them with appropriate options. Since the initial bootstrap, provisions have been added for dataset name substitution, so that the same program can be applied to datasets with different names. The usual way of creating output information is to build it up in a string by concatenation, using implicit conversions to convert numbers to strings. Then the string is written out as a whole. For input, a string is brought in and broken down into parts using the SUBSTR built-in function. Strings representing numbers can then be implicitly converted to the numbers themselves. Work is presently under way on providing stream transput facilities, which include both formatted and unformatted transput of lists of variables.

A limited set of on-conditions has been added since the initial version of CIS PL/I became operational. These were chosen mostly because they were needed by the compiler itself. Perhaps the most useful of these is ENDFILE, which allows the programmer to take some specified action when end-of-file is encountered on input from a specified dataset. Another, STORAGE, allows the compiler to change its field length dynamically.

At present the list of built-in functions is quite limited. Since many of these functions can be implemented using existing Fortran library routines, most of the work depends only upon providing appropriate linkages. It is possible to access the Fortran library even under the present implementation by calling a Fortran function, since calling sequences are compatible for scalar numerical arguments.

A number of implicit and explicit type conversions are in the present implementation; conversion to and from character strings has been emphasized. These conversions take place both on assignment and when arguments are passed to procedures. Aggregate arguments can be passed to procedures, though type conversion for such arguments has yet to be implemented.

2.2 Features Not Yet Implemented

The major features that are lacking are:

1. Stream transput. PL/I has three varieties of stream transput, none of which are currently implemented. Edit-directed transput is similar to Fortran format-directed transput but is far more dynamic. PL/I allows arbitrary expressions in most contexts where Fortran requires a constant, and the values of these expressions can be influenced by data being read in under control of the format statement. In the extreme case, an expression in a format statement could invoke a function which in turn would initiate another transput operation, perhaps even on the same dataset!

List-directed input causes data to be read in a free-field format, and among other things, requires that type conversions be selected dynamically during the input process. (For instance, a data item in floating-point representation might be read into a fixed variable.) List-directed output causes the value of a sequence of expressions to be written onto a dataset at fixed tab positions. Data-directed input and output associate the name of a variable with its value; in particular, on input the name of the variable being read in is part of the input. Thus a run-time symbol table must be kept.

2. Record transput. A very limited form of record transput is the only form of transput presently in CIMS PL/I. Record transput is used to achieve random access to secondary storage by means of keys, and requires a rather extensive interface with the operating system. Record transput also interacts with multitasking, since a record transput operation may overlap computation as well as other transput operations.

3. Aggregates. The full treatment of aggregate data (structures and arrays) is one of the thorniest aspects of PL/I implementation. Procedures should be able to return aggregates as their values, and aggregates should be able to be passed as arguments to built-in functions. Further, type conversion must

be implemented for aggregates, and this conversion may require the development of an aggregate from a scalar. PL/I provides the ability to reference cross-sections of arrays, but this ability is not yet in CIMS PL/I. Further, through the use of the DEFINED attribute, one aggregate may be defined in terms of a mapping from another aggregate. The DEFINED attribute has not yet been implemented either.

4. Arithmetic data types. PL/I allows three independent arithmetic attributes to be applied: base (DECIMAL or BINARY), scale (FIXED or FLOAT) and mode (REAL or COMPLEX). Furthermore, precisions (number of digits to be kept) and scale factors may be specified. Since at present all fixed numbers behave like Fortran integers, a great deal remains to be done in this area.

5. Multi-tasking. PL/I provides extensive facilities for the generation of parallel tasks and for task synchronization. These facilities are useful even on a machine with only one central processor, since tasks may be dependent upon transport operations. None of these facilities are yet implemented, though certain hooks for them exist.

6. Pictures. PL/I has a PICTURE facility much like that of COBOL. Pictures are useful for describing and editing data used in commercial applications. Pictures are not presently implemented.

7. Data alignment. PL/I provides the ALIGNED and UNALIGNED attributes to control the tradeoff between economy of storage and speed of access. Although on the surface it might appear that a correct implementation could ignore these attributes, it happens that they have certain implications with regard to the storage of character and bit strings. In particular, adjacent character strings in an unaligned structure must be stored adjacently even if that requires starting one of them in the middle of a word. A similar situation holds for bit strings. One implication of this requirement is that pointers must be able to point to bits or characters within a word; pointers as presently implemented in CIMS PL/I can only point to words.

8. Bit strings. The implementation of bit strings is presently quite restrictive, as they can only be of fixed length and that length must not exceed 60. Varying bit strings are not implemented at all.

9. Built-in functions. PL/I requires a rather large library of built-in functions, and in addition there are a number of implementation-defined ones that would be desirable. Only a few built-in functions have been implemented so far, though many of the remaining ones could be taken directly from the Fortran library.

2.3 Operating Characteristics

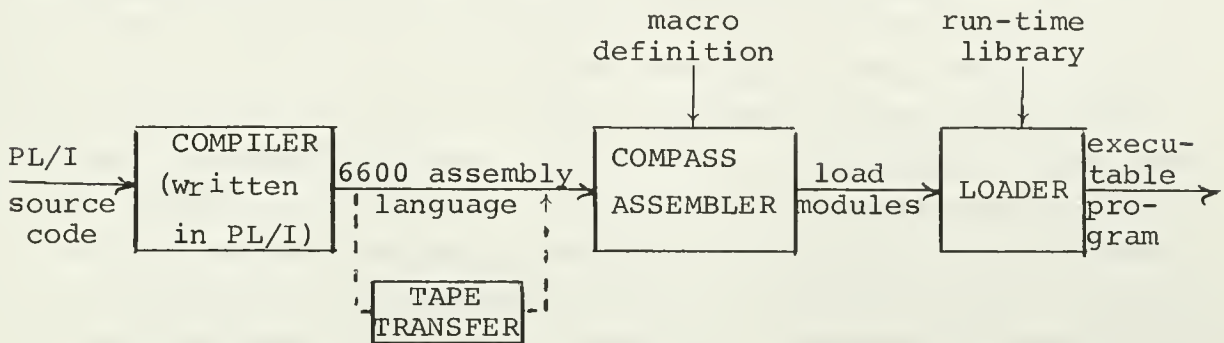
The compiler is designed to operate under the CDC Scope 3.2 Operating System [6] on the CDC 6000 computer series, though it will also run under the Scope 3.4 Operating System [7]. The input to the compiler is a sequence of PL/I external procedures, and the output is in the form of 6600 assembly language. This assembly language must then be assembled in a separate step, using a set of macro definitions provided with the compiler. Finally, the assembled program is loaded together with a run-time library and executed.

The compiler itself presently operates at a rate of about 15 statements per second and requires a minimum field length of 101K octal; large programs require considerably more space. Assembly takes about 30% more time than compilation, so that the total rate of compilation is about 6 statements per second.

An extensive set of compile-time diagnostics has been provided. Run-time diagnostics are much sketchier, though a recently added feature makes it easy to determine the point at which a program has stopped. Listings are unornamented except for the inclusion of statement numbers. A listing of variables and a cross-reference map can be obtained.

3. Bootstrapping Method

The CIMS PL/I compiler is written in PL/I and was bootstrapped from the IBM 360. In Figure 1 we see the process involved in obtaining an executable program from a source program. The source program is processed by the compiler to



Dotted path is taken during bootstrapping.

Figure 1. Compilation Process and Bootstrap Procedure

produce a version in COMPASS, the 6600 assembly language. Since the COMPASS version is expressed as lines of code, the compiler can be viewed as transforming character strings to character strings. (In this analysis we are ignoring such matters as output listings.) The COMPASS code consists mostly of calls on macros, so that the assembler must be provided with a collection of macro definitions. The output of the assembler is a collection of load modules, one for each external procedure. The compiled code will call upon various run-time subroutines for services such as storage allocation, character manipulation, and evaluation of built-in functions. These run-time subroutines are linked to the compiled code by the loader, so that the result of the loading process is an executable program.

Now let's return to the compiler itself, which we shall call C1. C1 is written in PL/I and transforms character strings

to character strings. Therefore it will run on any machine that has an operational PL/I compiler; we shall call this second compiler C2. Furthermore C1 is almost entirely machine-independent. There are a few machine dependencies such as the character tables used by the lexical scanner, the transformation used by the hashing algorithm, and the overlay calling conventions. These sections must be changed to correspond to the machine upon which C1 is executing.

When this project began, of course, PL/I did not exist on the 6600. However, we can consider a rather awkward way of obtaining executable PL/I programs on the 6600 under these circumstances. First, C1 is compiled on the 360 using C2. Since C1 is a PL/I program and C2 compiles PL/I programs so that they will execute on the 360, there are no difficulties here. The result is that C1 now executes on the 360, and converts PL/I source code to 6600 assembly language. It might seem that the process would fail since C2 turns out 360 machine language. Remember though that C1 is just a PL/I program that transforms character strings to character strings. The fact that the output character strings are 6600 assembly language is irrelevant.

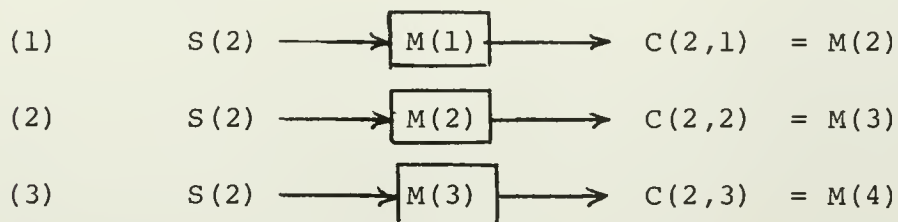
We now have our awkward method of compiling PL/I for the 6600. We take our PL/I source code, pass it through C1 on the 360, and write the results on tape. We then read the tape on the 6600 and carry out the rest of the process, as shown in Figure 1.

The bootstrap approach is based on the observation that C1 is itself written in PL/I, and therefore this awkward method of compilation can be applied. Thus, we pass the text of C1 through the process of Figure 1, including the tape transfer across machines. The result is C1 as an executable program on the 6600 -- in other words, an operational PL/I compiler on the 6600. From this point on the 360 is not used, and the dotted connection of Figure 1 is replaced by a direct connection.

Bootstrapping from the 6600 to the 6600 is necessary whenever improvements are made to the compiler. These improvements are of two sorts. Language extensions may enable the

compiler to do things that it could not do previously. For instance, the original version had no means of detecting end-of-file, and so the compilation of each external procedure required a separate job step. When the ENDFILE on-condition was added, the compiler could be modified to allow for batched compilations. Optimization of compiled code is an even more important advantage of repeated bootstrapping. As the code turned out by the compiler becomes more efficient, the performance of the compiler itself becomes improved through repeated bootstrapping.

An incidental advantage of bootstrapping is that it provides a severe test of the compiler itself. The approach is shown in Figure 2. We start off with an old machine-language version $M(1)$ and a new source-language version $S(2)$.



$M(i)$ machine language-version i
 $S(i)$ source language version i
 $C(i,j)$ machine-language compiler
 produced by compiling source
 version i under machine version j

Figure 2. Iterated Bootstrapping

The first compilation produces $M(2)$, which we then use to compile $S(2)$ again and produce $M(3)$. The process is iterated once more to produce $M(4)$. If the bootstrap process has succeeded, then $M(3)$ and $M(4)$ will be bit-for-bit identical,

a condition that can easily be checked mechanically. To see this, we note that the code generated by $S(2)$ should be independent of the C used to compile $S(2)$, since $S(2)$ as we noted before is just transforming character strings to character strings. Now $M(2)$ and $M(3)$ are both realizations of $S(2)$ and thus should transform any input, including $S(2)$ itself, in the same way. We now observe that if $M(3)$ and $M(4)$ are bit-for-bit identical, then any number of further iterations will produce results identical to $M(3)$. Thus we know that stability has been achieved. In typical practice, $M(1)$ performs the same transformation as $S(2)$ and hence as $M(2)$; in this case, $M(2)$ and $M(3)$ will be identical and the process can be terminated one step earlier. Since $S(2)$ may produce rather different code than $S(1)$, however, we cannot count on this early termination.

4. Structure of the Compiler

The major components of the compiler itself are shown in Figure 3. The input to the compiler is PL/I source code; the output is 6600 assembly language. The unit of processing is the external procedure, and each of the boxes shown represents a pass over part or all of the procedure.

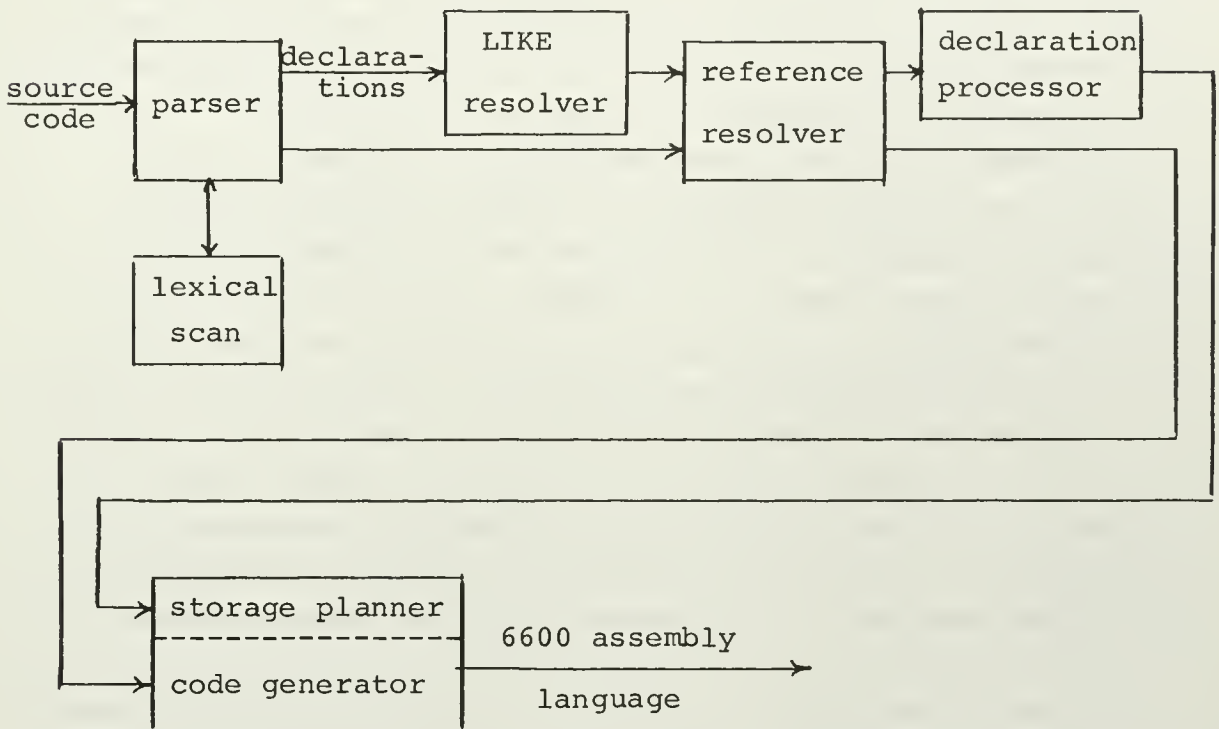


Figure 3. Structure of the Compiler

The first pass is performed by the parser, which is described in more detail in Section 5.1. It communicates with a lexical scanner which breaks down the input stream into tokens such as identifiers, numbers, and punctuators. The output of the parser is a collection of interconnected data structures. Some of these data structures are associated with declarations; most of the rest of them are associated with statements. The data structures

are interconnected through pointers, and their organization is discussed further in Section 5.2.

During parsing, a list is kept of all declarations that use the LIKE attribute. The LIKE attribute allows the programmer to copy a portion of a declaration into another declaration, and considerably simplifies the work of writing long but similar declarations. It must be processed at a very early stage because until it is processed the set of declarations is not complete. The parser keeps a list of all places where the LIKE attribute is used, so that it is not necessary to scan the entire text looking for occurrences of LIKE.

The task of the reference resolver is to connect uses of variables with their declarations. The reference resolver must scan through all statements as well as through all expressions appearing in declarations. Each use of a variable is represented by a pointer. This pointer is then replaced by a pointer to the appropriate declaration. The task of resolving a reference can be lengthy because of the complications introduced by block structuring and name qualification. Therefore different references to the same variable are represented by pointers with the same value, and the full work of reference resolution need be carried out only once for each distinct reference.

Reference resolution may turn up references to variables that are nowhere declared. Implicit declarations are generated for such variables. These declarations may require the application of contextual attributes, which are determined by the way that a variable is used. Contextual attributes are picked up during the parse and applied to implicit declarations during reference resolution.

The declaration processor generates the full set of declarations for the procedure. It must check and complete the declarations that have been given by the programmer, and it must also generate certain declarations that the programmer has not (and indeed, cannot have) specified. Completion of declarations is necessary because the attributes that the programmer has

written down may not be sufficient to completely characterize the declared variable. Completion of declarations also involves the application of any DEFAULT statements that the programmer may have written. When completing declarations we must also associate each built-in function with an index number. Each internal procedure must have an entry declaration generated for it, and subscripted labels have label array declarations generated for them.

The last and most complex pass is the code generator. The code generator processes the statements of the source program in the sequence in which they were written, except for 'DECLARE statements. The type of statement then determines the particular statement generator that must be called upon. Furthermore, whenever a BEGIN statement or a PROCEDURE statement is encountered, the storage planner is invoked. The storage planner is a major part of the code generator; its task is to develop the arrangement of storage that will exist during program execution.

The largest single component of the code generator is the procedure COMPUTE. COMPUTE receives an expression as its input. It generates code for evaluating that expression, and returns as its result a data structure that specifies the attributes of the computed value as well as the information needed to access that value.

5. Implementation Techniques

In this section I shall discuss a few interesting aspects of the implementation. This discussion is by no means exhaustive. It is intended to present a few selected issues and to show the approach that I took to them.

5.1 Parsing Method

The parsing method used in the compiler is detailed in [1], and will only be sketched here. It is based on a program called SYNDIPAR (SYNTAX Directed PARser), whose input is a specification of the required syntax and semantics and whose output is a PL/I parsing procedure together with the tables that drive it. SYNDIPAR itself is written in SNOBOL. In my early planning I assumed that SYNDIPAR would not have to be run very often and therefore its efficiency would be irrelevant. This has not turned out to be the case; the inefficiencies of SNOBOL have caused SYNDIPAR to consume inordinate amounts of resources, and I am planning to rewrite it in PL/I. Nevertheless, it works. (Of course, the compiler itself is in no way dependent upon SNOBOL.)

The first input to SYNDIPAR is a set of syntax equations for PL/I. These syntax equations are written in an extended form of BNF which includes operators for repetition both with and without delimiters between the items being repeated. For instance, an argument list is specified as a sequence of expressions delimited by, i.e., separated by, commas. SYNDIPAR starts with the assumption that the given grammar belongs to the class LL(1). The essential property of an LL(1) grammar is that at any point in the parse where alternatives exist, one can decide which way to go by looking only at the next symbol in the input string. Unfortunately PL/I does not have an LL(1) grammar. So the parser also must call upon "oracles" to look ahead in the input stream far enough to determine the correct decision in cases of doubt. We thus have a form of programming by exception. Since the situations where

oracles are needed are very few, we gain the efficiency of LL(1) parsing most of the time. The oracles themselves need not carry out a full analysis, but need only make a binary decision. They can do this by very simple means, such as looking for adjacent symbols of certain types or counting parentheses. An oracle need not concern itself with incorrect input; if the input is incorrect the oracle simply makes the most convenient decision and leaves it to other mechanisms to detect, announce, and possibly correct the error.

The syntax equations provided to SYNDIPAR also contain calls on semantic routines. The semantic routines are provided in the form of blocks of PL/I code, and their task is to generate the data structures that are the output of the parse. SYNDIPAR generates the appropriate linkages so that these blocks are called at the right time. The semantic routines are actually interspersed with the syntax equations, since it is easier to prepare the input in this form. The semantic routines are simply copied into the parsing procedure produced by SYNDIPAR, with some linkage material added front and back.

The first output of SYNDIPAR is the parsing procedure itself. Since almost all of this procedure is just a transcription of the input, there is not much to say about it. The second output is a sequence of instructions for a "parsing machine". The code for this machine is interpreted when an actual source program is being parsed. The interpreter itself is included in the parser. Each instruction for the parsing machine has an operator (there are ten of them) and an optional operand. Denoting the integer operand by x , we can sketch the operations:

1. Jump to the semantic routine at SLAB(x) , where the label SLAB(x) has been attached by SYNDIPAR to the appropriate block of code.

2. Jump to the syntax routine whose code starts with instruction x , saving a return address on a pushdown stack.

3. Unstack a return address and jump to it.

4. Use x together with the current symbol or symbol type to determine a branch address and then branch there. The

(x,symbol) pairs are used as input to a hashing algorithm. If no entry exists for the (x,symbol) pair then execute the next instruction.

5. Advance to the next token in the input stream.
6. Jump to the instruction at x.
7. Signal an error.
8. Plant a flag indicating that the currently governing error control routine is the block of code at ERR(x).
9. Not used.
10. There is an array of terminal symbols called TERM. If TERM(x) is equal to the next symbol, then skip one instruction; otherwise execute the next instruction.
11. If the symbol class (integer, string, punctuator, etc.) of the next symbol has code x, skip one instruction; otherwise execute the next instruction.

The use of oracles requires some special techniques in the lexical scanner. An oracle works by looking ahead through the input stream, searching for enough information to make its decision. It sees the input stream as a sequence of tokens rather than as a sequence of characters. These tokens will need to be scanned again during the subsequent syntactic analysis; furthermore, the same token sequence may be examined by several different oracles prior to analysis.

These requirements are met by providing two modes of lexical analysis. In the lookahead mode, scanned tokens are saved on a queue. Upon completion of lookahead, the current token is taken to be the one that was current when the lookahead started. In either lookahead or ordinary mode, tokens are taken from the queue until it is exhausted; thereafter they are taken from the input stream itself. In ordinary mode, scanned tokens are discarded; in lookahead mode, they are saved.

What if the space on the queue is exhausted? This situation will occur if an oracle is unable to make a decision after scanning some reasonable number of tokens. When the queue space is exhausted, the oracle is notified, and it makes an arbitrary

decision. If this decision is wrong, a syntactic error will be introduced artificially. Such errors are unavoidable if we don't want to store two unbounded alternative parses, since there are examples where the syntactic type of a construct remains indeterminate for an unbounded number of symbols.

5.2 Data Structure Design

The design of the data structures for the compiler was the most difficult and critical part of the overall design. Once the data structures were selected, most of the programming became straightforward. It is worthwhile examining some of the ideas underlying the data organization, and some of the tradeoffs that had to be considered.

5.2.1 Choice of Representation Method

The first decision concerning data structures was the choice between a syllable-oriented and a pointer-oriented representation. In a syllable-oriented representation, the program is encoded as a linear sequence of syllables. Syllables may either be type codes which determine the meaning of the following information, or representations of the information itself (including pointers and indices). The usual reverse-Polish representation is syllable-oriented, as are representations in terms of triples or quadruples. These representations have the convenient property that they can be written onto secondary storage, rewound, and reprocessed. Even the rewind can be avoided through clever tricks, though for disks this consideration is irrelevant.

In a pointer-oriented representation, sequencing is accomplished by using chains of pointers rather than by placing items one after the other. In a pointer-oriented representation the entire intermediate representation of a program must be directly addressable. For entities such as expressions the complexity of the two representations is about the same. Pointers, however, have the advantage that information need not be processed in the same order as it is generated.

It would seem that the case for a syllable-oriented representation is very strong, and for most programming languages that would be the natural choice. PL/I, though, poses some special problems, most of which arise from the fact that declarations may be scattered throughout a block rather than all appearing at the head of a block. Thus the code associated with declarations must somehow be moved to the right place, and this kind of code motion over an arbitrary distance becomes rather complex in a syllable-oriented scheme. For instance, a block could conclude with a declaration of a character string whose length is given by an expression. Somehow the length expression must be moved to the start of the block, since it must be evaluated before execution of the block itself can begin. A similar problem occurs with implicitly declared variables. Even if the programmer writes all his declarations at the head of the block, there still is the problem of procedures internal to the block, for which declarations must be generated, as well as the problem of detecting and dimensioning constant label arrays.

It is apparent that the syllable approach is not very helpful for representing declarations. Declarations are by nature interlinked in many ways, and simply do not have the sort of sequential structure necessary for representation by syllables.

A further difficulty with the syllable-oriented approach arises in the analysis of expressions. A simple bottom-up analysis is not sufficient if certain kinds of local optimization are to be attempted, since one wants to consider an operand in conjunction with the operator being applied to it. In a reverse-Polish representation it is awkward to locate such an operator before both operands have been analyzed.

In the original bootstrap version of the compiler the objective was to get it on the air as soon as possible. I therefore chose to use the pointer-oriented approach primarily because it was easier to implement and did not require any transput facilities. In retrospect the decision is still not

clear-cut, though there are places where syllables could have been used to good effect. The tradeoffs are influenced by the availability of memory, either virtual or real. These considerations are discussed in Section 6.3.

5.2.2 Examination of Tradeoffs for Specific Structures

As an example of tradeoffs, consider the representation of identifiers within the data structures. An occurrence of an identifier might be represented by the character string for the identifier or by a pointer to some other data structure that contains that character string. In examining the space requirements we see that if most identifiers are short (as we would usually expect them to be), then it is more economical to store the character strings directly. However, there is a trap. Since identifiers vary in length, we must store not only the character string but also its length. At this point the economy of space begins to favor the indirect representation. The indirect representation has another cost: the hashed table search necessary to locate the unique structure associated with a given identifier. However, this cost must be paid eventually anyway since it is necessary to hash identifiers in order to resolve references. The compiler uses the indirect approach. Every identifier has a unique structure that represents it. The identifier is characterized by a pointer to this structure. Associated with the structure is a list of all declarations of that identifier. The list is formed by pointing to the first such declaration and then chaining the rest of them through a pointer component of the declaration itself.

As another example, consider the structure that is used to describe the declaration of a single variable. The actual PL/I specification of that structure is given by:

```

DECLARE 1 DECL BASED(DPTR) ,
  2 ATTRIBS BIT(60) , /* BINARY ATTRIBUTES */
  2 COMPLEX_ATTRIBS POINTER, /* POINTER TO LIST OF COMPLEX
      ATTRIBUTES */
  2 ACCESS_RULE POINTER, /* POINTER TO STRUCTURE FOR ACCESSING
      LOCATION OF THIS VARIABLE */
  2 PARENT POINTER, /* CONTAINING STRUCTURE OR BLOCK */
  2 SIBLING POINTER, /* NEXT VARIABLE ON SAME LEVEL */
  2 LINK POINTER, /* CONTINUATION OF LIST OF DECLARATIONS
      OF SAME IDENTIFIER */
  2 IDENTIFIER POINTER, /* STRUCTURE CONTAINING NAME OF
      IDENTIFIER BEING DECLARED */
  2 STMTNO FIXED BIN, /* NUMBER OF STATEMENT DECLARING
      THIS VARIABLE */
  2 NEXTLEVEL FIXED BIN; /* LEVEL OF STATIC NESTING OF
      THIS VARIABLE */

```

The name of the entire structure is DECL, and its components are ATTRIBS, COMPLEX_ATTRIBS, etc. In designing this data structure, space and ease of access must be traded off. The problem is that while most declarations of variables are quite simple, the most complicated ones are very complicated indeed. To represent all possibilities directly would lead to a very large structure for even the simplest sort of declaration. The details of DECL suggest how the tradeoff was handled.

Consider, first, the representation of attributes. The declaration of a variable may attach to that variable a variety of attributes. Some of them may be recorded by a single bit, e.g., EXTERNAL. Others may require one or two integers, e.g., PRECISION. Others require an expression, e.g., CHARACTER, though the obvious representation of an expression here is as a pointer to something else. Yet others require a complex ad hoc structure, e.g., ENTRY or INITIAL.

The DECL structure uses two different devices for representing attributes. First there is a bit string, ATTRIBS, in which one bit is devoted to representing the presence or absence of each attribute. Second, there is a pointer COMPLEX_ATTRIBS to a linked list of complex attributes. Each of these includes a flag to indicate its type, a pointer to the next complex attribute, and further material determined by which attribute is being described. There are, of course, further data

structures, not given here, for describing each attribute type. It is convenient to include a bit in ATTRIBS for attributes that also appear in the list of complex attributes.

For a declaration with no complex attributes, then, the declaration contains only one extraneous piece of information: a null pointer as the value of COMPLEX_ATTRIBS. For more elaborate declarations we will have correspondingly elaborate lists of complex attributes. Note that the complex attributes are not necessarily mutually exclusive. For instance, a variable could have the BASED, DIMENSION, INITIAL, and CHARACTER attributes simultaneously.

A similar economy issue is raised by the fact that most PL/I variables are unstructured, but yet we must be able to represent structured variables. To represent an element of a general (i.e., not necessarily binary) tree we need to know the parent, first child, and next sibling of that element as well as any information associated with the element itself. Structured variables do indeed have the organization of a general tree. Yet an unstructured variable needs none of this tree-oriented information.

We note, however, that for every variable we must be able to identify the block in which this variable is declared. We achieve this by treating the parent of a level-one variable (i.e., one with no parent) as being the descriptor of the declaring block. Thus we no longer have orphan variables; every variable has a functional parent, and the PARENT component of DECL will always be put to good use. We can, of course, find the block associated with any variable not at level-one by tracing up the chain of ancestors.

We also note that for every block we will want a list of variables declared in that block. We can form that list by stringing together all the level-one variables; any other variable can be found by tracing downwards through the trees. Since a level-one variable has no sibling, the SIBLING component of DECL can be used for this purpose. Not all variables will

have siblings under this interpretation, but most of them will; and that is sufficient to justify the use of space.

How about the first child? Unlike the two other tree links, the first child has no obvious alternate use for variables that don't have a first child. And most variables do not in fact have children. So we do not set aside a component of DECL for the first child; rather, we store this information as a complex attribute, and spend the space only when it is really necessary.

These examples may illuminate the nature of the decisions that have to be made. We are not worrying about the detailed layout of bits in storage, but rather about the more general tradeoff of indirection versus space. If information is stored explicitly wherever it is needed, then it will take more space. If it is never stored explicitly when a path to it exists, then processing it will take more time, and furthermore will lead to more complex programming. Some implications of this kind of decision for programming languages are discussed in Section 6.1.

5.3 Run-Time Storage Allocation

Run-time storage allocation is carried out through the use of a generalized block allocator. The allocator has the ability to obtain a block of storage of given size, or to free a given block. An allocated block is never moved. If a block is freed and either of its neighbors is freed, then the adjacent free blocks are combined to form a larger block.

Storage may be required either as a consequence of entering a procedure, evaluating an expression, or executing an ALLOCATE statement. The treatment of ALLOCATE statements is straightforward; the necessary amount of storage is determined and the allocator is then asked for a block of that size. The block is retained until a FREE statement for it is executed.

The remaining cases involve storage requests that are not explicitly made by the programmer. We may classify storage requests according to the time at which the size of the request becomes known. This time may be at compile time, at block entrance time, or an expression evaluation time. For the moment,

assume that we use a pushdown stack. Upon procedure entrance, we set aside an amount of stack space obtained by adding the amount of header information to the amount of storage required for variables and temporaries whose size is known at compile time (excluding STATIC variables, which are allocated separately). We then calculate the amount of storage needed for variables and temporaries whose size is known at block entrance time, and set aside that further storage on the stack. When evaluating an expression we may need temporary storage (though not storage for variables) where the amount of storage needed cannot be determined until expression evaluation. Such storage is presently not allocated on the stack, but rather is obtained directly from the block allocator.

Because of the requirements of multitasking and the ALLOCATE statement, a straight pushdown stack storage allocation regime is not feasible. A modified approach, which I like to call "sticky allocation," has been used. Space for the pushdown stack is obtained in large but noncontiguous chunks. These chunks are linked together. Within a chunk, allocation proceeds in pushdown fashion. When a chunk is exhausted, a new chunk is obtained from the allocator. The size of the new chunk is determined by adding a constant to the amount of pushdown space needed. When the stack retracts, however, a vacant chunk is not freed immediately. Instead, a buffer of one vacant chunk is retained. Thus even if one encounters a series of procedure calls at a point where a chunk is just filled up, the overhead of obtaining a new chunk from the allocator can be avoided.

The difficulty with storage allocation at present is that the internal organization of the stack is unduly complicated. Furthermore, all temporary storage really should be placed on the stack. The consequence of the present arrangements is that prologues and epilogues, which must be performed upon procedure entrance and exit, are both complicated and time-consuming.

6. Conclusions

In this section I shall present some of the conclusions that I drew from this work.

6.1 Sources of Program Complexity

The source language version of the compiler occupies about 6000 card images, and constitutes a large and complex program. It is fruitful to inquire as to the causes of this complexity.

As a program evolves from an idea to a reality, conventions and concepts are tentatively set forth and then modified. For instance, data structures are designed with algorithms in mind; specification of the algorithms then shows up flaws in the data structures. Thus the program production process involves continual change. Each decision is bound by decisions made in the past, and influenced by the anticipation of decisions yet to be made. An analogy can be drawn with filling in a crossword puzzle. If you are filling in a line and are not sure which of two words fits at that point, you will look at the cross definition to make the more plausible choice. Even if you do not know the cross words you will have some information about them, e.g., that certain letter sequences are unlikely to occur.

Present decisions can also influence past ones. When conceptual flaws are discovered, past decisions may need to be modified. These modifications, in turn, will propagate and so their consequences must be accounted for. Thus the case for breadth-first programming versus depth-first programming is not clear cut. In either case one starts with an overall goal and breaks it up into a series of subgoals $G_1, G_2, \dots, G_n$. With depth-first programming, one does all of G_1 before proceeding to G_2 . In breadth-first programming one does G_1 through G_n to another level of detail without fully expanding any of them. In depth-first programming one is likely to program G_1 without having designed many details of G_2 . This practice has the disadvantage of not anticipating relevant characteristics of $G_2, \dots, G_n$. It has the advantage, though, that in creating $G_2, \dots, G_n$ one knows all about G_1 .

One cause of complexity, then, is the propagation of decisions. It follows that automation of this propagation will lead to a reduction of complexity. I will give an example of how automatic propagation of decisions has already been used to reduce complexity. The data structures used by the compiler are specified through PL/I structure declarations. These declarations are kept as so-called "common decks", recognized by the CDC UPDATE program. Only one copy is kept, and each procedure references those declarations that it needs. A change to this one copy will then propagate to all procedures that use the declaration.

The most frequent situation is that we need to add a component to a data structure. In this case the propagation is quite automatic. In the case where the meaning of a component is changed, no automatic propagation is possible, though automatic warning through some cross-referencing scheme might still be possible. This case is worth examining in more detail.

Automatic propagation of changes of meaning is best served by factoring out the definition of meaning from the use of that meaning. I shall give some examples. Frequently a variable can assume some set of meaningful values, e.g., "red", "blue", "yellow". If we encode these three values as 1, 2, and 3, then changes in the encoding will force changes in the programs that use it. If the names of the colors themselves are used, then the program becomes independent of changes of representation. So what we want is a declarative facility to associate names with colors. Such a facility is actually provided in COBOL and in Algol 68, but not in PL/I. Another example has to do with the association of a property with an object. Such an association might be made either by storing the property as part of the object or by having the property computed by some function. It would be desirable to factor out this decision by using a functional notation uniformly and then letting the function definition reduce to either a subroutine call or an appropriately indexed address reference.

There is a view currently popular among the advocates of "structured programming" that all data structures should be hidden by procedure calls. In this view, access to a data structure should be obtained by calling an appropriate procedure. This approach does have advantages vis-à-vis access control, but it transfers rather than solves the problem of changes of meaning. Data structures in an application such as compilation are necessarily highly interlinked, and so the value returned by a procedure will often have its meaning determined by some underlying data structure. For instance, a binary choice, represented by a predicate, may turn into a three-way choice.

A related problem stems naturally from the general graph structure of typical programs, in which procedures are called from many different places. It often turns out that the different calling points each requires a slightly different service or piece of information. This gives rise to a procedure organization that resembles a "theme with variations". In general, the necessary variations on the theme will not all be apparent at the time when the procedure is designed. Often the multiple-use aspects of such procedures are not at all apparent at the time the procedures are written. Top-down programming does not solve the problem because recursion may give rise to cycles in the program graph.

An example of a theme with variations is the procedure LOCN, which is part of the code generator of the compiler. The task of LOCN is to calculate the location associated with a reference to a variable. The reference may or may not be subscripted and may or may not be based on some pointer. Subscripts may or may not be implicit. For instance, if A is an array the expression "ADDR(A)" should retrieve the location of the array itself, but the statement "A=l;" should retrieve individual elements of A chosen by implicit subscripts. Thus the use of LOCN depends on the context. There are two different entry points to cover the two cases, plus a third entry point to cover the case where an offset from a base is desired rather

than an actual location. All three cases share most of the same code; but there are various internal case-dependent choices that give rise to the organization of the theme with variations.

6.2 Operating Systems Support

During the development of the compiler I used two different operating systems: OS/360 on the IBM 360 and Scope 3.2 on the CDC 6600. The support offered by these systems became a critical issue during this development. Two quite different kinds of support were involved: support of the programming process itself, and run-time facilities available to the compiler.

The PL/I compiler exists in three different forms: as a collection of symbolic source programs, as a collection of compiled load modules, and as a collection of absolute overlays. All three forms require maintenance, but the symbolic one is the most critical. File maintenance was a very time-consuming operation, particularly under OS/360 where the facilities are grossly inadequate. IEBUPDTE, the symbolic updating program under OS/360, is difficult to use because of unclear specifications, arbitrary restrictions, and an error-prone input form. Under Scope 3.2, on the other hand, I was able to carry out file maintenance expeditiously and with a minimum of fuss. My success was due to the combined use of the permanent file system for saving programs, the CDC symbolic update program UPDATE, and the NYU interactive editor designed by Richard Reich. Under both operating systems I found that it took some time to develop adequate file maintenance procedures, but that once these procedures were developed their use saved a great deal of time.

Adequate editing facilities are essential in the reduction of program complexity. There seems to be no substitute for interactive editing in this matter. The evolutionary nature of complex programs assures that reorganizations will be necessary, and during these reorganizations one wants to be able to move blocks of text from one place to another as well as to

carry out various global substitutions. The ability to move large text blocks is a facility often lacking in editors. Reformatting is also important, though this may not be an operating system function because of its language dependence.

A good measure of the adequacy of programming support has to do with the proportion of runs that are lost because of errors outside the program being debugged. Such errors include job control language, problems, compiler errors, updating problems, and machine troubles. As I kept no records, my impressions must be subjective; but I would estimate that overall I lost perhaps two-thirds of my runs on the 360 for system-related reasons and perhaps one-tenth of my runs on the 6600 for system-related reasons. The comparison is somewhat unfair to the 360 because much of my work on the 6600 was done interactively; nevertheless, "battling the system" was a major part of my effort in getting the bootstrap going.

Neither the 360 nor the 6600 provided adequate debugging facilities. On the 360 certain facilities were built into the compiler itself, e.g., the CHECK condition, but their use required modification of the source program; and IEBUPDTE provided no systematic way of removing these changes once made. Debugging in machine language on the 360 was impossible since I did not have the requisite knowledge of the IBM compiler or of OS/360. On the 6600 I used a machine-language interactive debugger developed by Peter Maclean [4], which was the best that could be hoped for under the circumstances. Some PL/I source language debugging facilities that don't require recompilation are presently being developed by Janet Fabri as a master's thesis.

The compiler required run-time support for overlaying. The linkage editor on the 360 provided better support for overlaying than any corresponding facility under Scope 3.2 on the 6600. The compiler is now organized into overlays, but this had to be done in a machine-dependent manner. The loader under the Scope 3.4 operating system for the 6600 does have facilities as good as those of the 360 linkage editor.

Another area where run-time support is needed is in error recovery. The facilities on the 360 for error recovery are superior to those on the 6600, though some improvement can be expected on the 6600 when Scope 3.4 comes into operation. On the 360, it is possible for the program to regain control from machine interrupts such as overflow or reference out of bounds. On the 6600 control is taken away from the program under such circumstances. Neither machine provides program recovery from time limit, though a restricted facility for time limit recovery is provided by Scope 3.4 in the form of "reprieves" for various errors. It would be very helpful if the programmer could specify a local time limit, not to exceed his global time limit, and be interrupted when this time limit was reached. With such a facility the programmer could use the time limit for resource allocation, and could perform as much processing as he wished to following the exhaustion of a local time limit. Since the global time limit would still be inviolate there would not be any adverse impact on job scheduling.

Run-time support is also needed for transput and for the mathematical library. As these facilities in CIMS PL/I are very limited, I did not have much experience with the adequacy of Scope 3.2 in this regard. The issue did not arise in my work under OS/360.

6.3 Virtual Memory

My experience suggests some compelling reasons why compilers can function more effectively in a large address space such as that usually provided by a virtual memory. Various decisions about compiler design are dependent on the availability of address space. The situation shows up in two areas: program structure and data structure. A small address space dictates the use of overlays in program organization, and can lead to compilers with a great number of specialized phases, e.g., the IBM F-compiler. The resulting program organization is quite complicated and furthermore generates a great deal of transput

activity. With a large address space one can ignore the overlaying problem. If only a small physical memory is available, then the paging processes of the operating system will produce the effects of overlaying without the corresponding complications in the program. Paging will still require transput activity, but that activity would be necessary in any case. If physical memory is large, then the transput activity will be avoided since little paging will be necessary. Thus a single program organization can function optimally over a wide range of availability of physical memory. Good programming practices should still be followed to preserve locality of reference. Some experiences with the MULTICS PL/I compiler, though, suggest that some phase-oriented organization may be desirable even in a virtual memory environment.

The question of syllable-oriented intermediate representation versus pointer-oriented intermediate representation is much less vexing in a large address space, since one can use that representation which is the most natural at each point (cf. Sec. 5.2.1). The entire intermediate representation becomes addressable, and thus the problems of code motion alluded to previously are no problem at all. The virtual memory lends itself naturally to coping with a wide range of source program sizes; the amount of paging activity becomes proportional to the program size. For short programs the intermediate representation often will not be paged out.

6.4 Reliability

In order to maintain the reliability of the compiler I developed a series of acceptance tests. The most stringent test is to have the compiler compile itself as described in Section 3; success at this point almost certainly implies that any later errors found in the compiler can be fixed. There is a danger that must be avoided: if an error in the compiler makes it impossible to recompile the code section containing the error, then there is no way to escape except by returning to a previous safe version. Since it is expensive to recompile the entire

compiler I have done it only about every two months. Because of the range of language facilities used by the compiler itself, the self-compilation test has turned out to be quite effective.

Whenever any change at all is made to the compiler or to its run-time library, I check it by running a standard acceptance test. The first version of the acceptance test was designed to run through all code paths in the compiler except for those associated with errors. Thereafter, whenever any error was discovered either in the compiler or in its library, a further test case was added to cover that error. Thus the acceptance test provides warning if any previously found error recurs. Not surprisingly, most of the errors that have shown up occurred when the compiler aborted as a result of a programmer error.

In the design of an acceptance test, the output should be as short as possible. Only incorrect results should produce messages. Thus one need not scan a long output to see if it is the same as the previous one, a process which is prone to oversights. The present acceptance test produces only one line of diagnostic output if the compiler is working properly, though it does produce a listing of the testing program. Any malfunction is evident at a glance.

6.5 Project Organization

Up to the completion of the first bootstrap, the compiler was essentially a one-person project. The obvious disadvantage of such an organization is the limit on how much code one person can produce. A more subtle disadvantage is the lack of interaction among co-workers that often leads to improvements in design.

On the other side, one-person programming seems to avoid most of the interface difficulties that plague multi-programmer projects. It eliminates the misunderstandings that often make it hard for modules written by different people to communicate with one another. The "chief programmer team" approach suggested by Mills and described by Baker [2] seems to preserve the spirit of one-person programming while eliminating many of its

disadvantages. My experience suggests that additional help can be best utilized when two criteria are satisfied:

- (1) the task to be done is unambiguously defined, and
- (2) the task to be done is not on the critical path.

The best way to define a programming task unambiguously is to pose it in the form of an addition to a program that is already working. The existing program then defines the necessary interfaces. Additional help is often useful, too, in gathering information. For instance, operating systems facilities are often poorly documented, and obtaining information about them is a task that can usefully be farmed out.

6.6 PL/I as a Systems Programming Language

One of the original design goals of PL/I was to provide an adequate language for systems programming. It has been used as the systems programming language at MULTICS [3]. Experience with the compiler suggests that while PL/I is probably better than most other languages around for that purpose, it is far from ideal.

The strongest argument for PL/I is the wide range of facilities that it provides. The description of CIMS PL/I indicates the range of facilities that were actually used in writing it. Some examples are: different forms of storage allocation, pointer manipulation, bit strings and character strings and the associated built-in functions, structured variables (particularly valuable), block structure and recursion, and flexible forms of iteration. Another argument for PL/I is that usually one can accomplish much by writing down little; the language has great expressive power.

There are, of course, failings of PL/I. The pointer structure is such that it is impossible to write any reasonable form of garbage collector; the list processing facilities require far too much of the user. The iteration structure, while flexible, still does not allow some more recently suggested concepts that lead to cleaner programming, e.g., specifying a loop as a cycle and placing an escape clause at an arbitrary place within it.

Structuring should be associated with data types rather than with variables; the approach of Algol 68 in this area seems far more sensible. It would be desirable, in fact, to use a functional representation for structures and then to be able to define the function later as being either computed by some procedure or found merely by address manipulation. The PL/I on-conditions seem unnecessarily general, and often do not provide simple but desirable effects. For instance, storage allocation in PL/I can be carried out in specified areas. When an area is exhausted, an on-condition is raised to signal this fact. However, there is no way to determine which area was the one that was exhausted, except through knowledge of the point of execution of the program. The obvious solution, writing an allocation function, does not work since although one can pass allocated storage to functions, one cannot pass data descriptors to them in any reasonable way.

Efficiency is certainly an issue, but the significance of it is not yet known. It seems clear that PL/I compilers can be far more optimized than they have been so far. My own view is the inefficiency in generated code can be brought to a sufficiently low level so that it is not a major consideration.

I do not regard the size of PL/I as a disadvantage for systems programming. A reasonable implementation should not include run-time support for a given program beyond that which it actually requires. Thus a program is not penalized for those facilities that it does not use.

6.7 Design Flaws

Naturally, experience in coding and using the compiler has shown up a number of flaws in the original design. It was possible to correct some of them without great difficulty, but others will require extensive rewriting.

To start with the bright side, I will describe some of the flaws that have been discovered and corrected. The inability to detect end-of-file and to rewind files was a serious lack in

the original version. This lack was remedied by adding various on-conditions, including ENDFILE, and also by adding an environment option that allows files to be opened or closed with or without rewind. The original version was unable to compile more than one external procedure on a single call. Correction of this flaw required the insertion of some control mechanisms that could not be written in PL/I because of their extralingual manipulation of the storage allocation state. In the original version, error diagnostics were separated from the procedures to which they referred, and this situation made it difficult to correlate error complaints with statements. Finally, a small change was made so that one can determine the statement being executed at the time of an error. This is done by setting machine register A0 to the current statement number at the beginning of execution of each statement. (The user can turn off this mode of compilation if he wishes.) Since the standard dump produced on an error gives the value of A0, it is not difficult to isolate the offending statement.

Serious flaws, of course, remain. The worst of them is that the code generator directly produces register-dependent code. Keeping track of registers has led to great complexity. The register-dependence of the code also makes optimization very difficult and inhibits adding a number of additional language features. An unexpected problem was that the heavy use of macros and micros caused the 6600 assembler to use an excessive amount of time. It was surprising to discover that a compilation actually takes less time than assembly of the resulting code. The right approach seems to be to produce some intermediate code form that is register-independent and perhaps even machine-independent, and to add a further pass to convert this code to register-dependent instructions and macro instructions, with expansions kept as simple as possible.

Another problem is the constraints imposed upon the parsing procedure by SYNDIPAR. Because the parse must be written as one enormous procedure, it is not possible to use recursion

within the semantic routines of the parser. The result is that the parser is unnecessarily complex, and also uses various tables whose sizes cannot be intelligently determined in advance. The difficulty is not fundamental, and could be cured by changing the conventions for calling semantic routines.

There are also problems in the way that pushdown stack storage is handled. These problems are discussed briefly in Section 5.3.

References

1. Abrahams, P., A syntax-directed parser for recalcitrant grammars. Intern. J. Computer Math., Sec. A, Vol. 3, 1972, pp. 105-115.
2. Baker, F. T., Chief programmer team management of production programming. IBM Systems J., Vol. 11, No. 1, 1972, pp. 56-73.
3. Corbato, F. J., PL/I as a tool for systems programming. Datamation, May 1969, pp. 68-76.
4. Maclean, Peter, An interactive debugging aid for the CDC 6600, M.S. Thesis, Computer Sci. Dept., New York Univ., February 1973.
5. PL/I BASIS/I-9. European Computer Manufacturers Association and American National Standards Institute, December 1972.
6. Control Data 6400/6500/6600 Computer Systems - SCOPE Reference Manual, Version 3.2. Publication No. 60189400, Control Data Corp., Palo Alto, Cal.
7. Control Data Cyber 70 Computer Systems - SCOPE Reference Manual, Version 3.4. Publication No. 60307200, Control Data Corp., Palo Alto, Cal.

Appendix A

CIMS PL/I Users Manual

 *
 * CHANGES AND NEW FEATURES *
 *

A, 4/2/73

1. THE INPUT-OUTPUT FACILITIES HAVE BEEN GREATLY IMPROVED. THE CHANGES INCLUDE THE ABILITY TO REWIND FILES, THE REMOVAL OF THE RESTRICTION THAT PERMANENT FILES CANNOT BE READ DIRECTLY, AND THE AVAILABILITY OF THE ENDFILE AND UNDEFINEDFILE CONDITIONS. SEE SECTIONS 2.1,7, 2.3.5, 2.3.16, AND 2.8 FOR DETAILS.
2. THE ON-STATEMENT IS NOW AVAILABLE. SEE SEC. 2.3.14 FOR DETAILS.
3. THE ALLOCATE STATEMENT NOW ALLOWS MULTIPLE ALLOCATIONS.

B, 4/23/73

1. THE COMPILER NOW EXPECTS ITS INPUT ON THE FILE -INPUT- AND PRODUCES THE SOURCE LISTING ON THE FILE -OUTPUT- (CF. SECS. 3.2, 3.3).
2. THE FILE -ERRORS- IS NO LONGER USED.
3. FILE SUBSTITUTION NOW RECOGNIZES NULL FILES (CF. SEC. 3.4.1).

C, 6/15/73

1. COMPILATIONS MAY NOW BE BATCHED, SO THAT ANY NUMBER OF EXTERNAL PROCEDURES CAN BE COMPILED WITH A SINGLE CONTROL CARD. AS THE COMPILED CODE IS NOW IN A SINGLE RECORD, THE RECMRG STEP IS ALSO ELIMINATED. THESE CHANGES MAKE THE CONTROL CARDS MUCH SIMPLER. FURTHERMORE, THE FILE -TABLES- IS NO LONGER NEEDED, AS IT HAS BEEN INCORPORATED INTO THE COMPILER PROPER. SEE SEC. 3.2 FOR DETAILS.
2. AUTOMATIC FIELD LENGTH MANAGEMENT HAS BEEN ADDED TO THE COMPILER. A COMPILATION LISTING NOW INDICATES THE FIELD LENGTH REQUIRED FOR COMPILATION.
3. A DEFAULT STATEMENT HAS BEEN ADDED. SEE APPENDIX A FOR DETAILS.
4. A CROSS-REFERENCE MAP CAN NOW BE OBTAINED. SEE SEC. 3.3 FOR DETAILS.
5. DUE TO A BUG THAT IS RATHER DIFFICULT TO FIX AT THE MOMENT, ON-UNITS CANNOT DECLARE ANY LOCAL AUTOMATIC VARIABLES. TO BYPASS THIS RESTRICTION, WRITE THE ON-UNIT AS A PROCEDURE AND CALL IT.

D, 6/27/73

1. A DEBUGGING OPTION HAS BEEN ADDED TO ALLOW DETERMINATION OF THE STATEMENT BEING EXECUTED AT THE TIME OF A PROGRAM ABORT. SEE SECS. 3.3,2 AND 4 FOR DETAILS.

TABLE OF CONTENTS

1. INTRODUCTION

2. LANGUAGE FEATURES

2.1 ATTRIBUTES

2.1.1 SCALAR DATA TYPES

2.1.1.1 FIXED

2.1.1.2 FLOAT

2.1.1.3 CHARACTER

2.1.1.4 BIT

2.1.1.5 POINTER

2.1.1.6 AREA

2.1.1.7 LABEL

2.1.1.8 ENTRY

2.1.2 AGGREGATION

2.1.3 STORAGE CLASSES

2.1.4 SCOPE ATTRIBUTES

2.1.5 INITIAL ATTRIBUTE

2.1.6 ENTRY ATTRIBUTE

2.1.7 FILE ATTRIBUTES

2.1.8 BUILTIN ATTRIBUTE

2.1.9 LIKE ATTRIBUTE

2.2 ATTRIBUTE DETERMINATION

2.3 STATEMENT TYPES

2.3.1 ASSIGNMENT STATEMENT

2.3.2 ALLOCATE STATEMENT

2.3.3 BEGIN STATEMENT

2.3.4 CALL STATEMENT

2.3.5 CLOSE STATEMENT

2.3.6 DECLARE STATEMENT

2.3.7 DEFAULT STATEMENT

2.3.8 DO STATEMENT

2.3.9 END STATEMENT

2.3.10 ENTRY STATEMENT

2.3.11 FREE STATEMENT

2.3.12 GO TO STATEMENT

2.3.13 IF STATEMENT

2.3.14 NULL STATEMENT

2.3.15 ON STATEMENT

2.3.16 OPEN STATEMENT

2.3.17 PROCEDURE STATEMENT

2.3.18 READ STATEMENT

2.3.19 RETURN STATEMENT

2.3.20 STOP STATEMENT

2.3.21 WRITE STATEMENT

2.4 EXPRESSIONS

2.5 TYPE CONVERSION

2.6 FUNCTION CALLS AND PARAMETER PASSAGE

2.7 BUILTIN FUNCTIONS

2.8 INPUT-OUTPUT

2.9 CHARACTER SET

2.10 KEY WORDS AND PHRASES

3. OPERATING ENVIRONMENT

3.1 INPUT FORMAT

3.2 CONTROL CARD SETUP

3.3 COMPILER OPTIONS

3.3.1 XREF

3.3.2 ST,NST

3.4 PROGRAM CALLING CARD

3.4.1 FILE NAME SUBSTITUTION

3.4.2 PASSING AN ARGUMENT TO A MAIN PROCEDURE

4. ERROR MESSAGES AND DEBUGGING TECHNIQUES

5. AREAS OF INCOMPATIBILITY

5.1 EXTENSIONS IN CIMS PL/I

5.2 INCOMPATIBILITIES BETWEEN CIMS PL/I AND IBM PL/I

6. FUTURE PLANS

7. ACKNOWLEDGMENTS

APPENDICES

A. THE DEFAULT STATEMENT

1. INTRODUCTION

AN EXPERIMENTAL PL/I COMPILER KNOWN AS CIMS PL/I IS NOW AVAILABLE ON THE 6600 AT THE AEC COMPUTING CENTER. I WELCOME ANY QUESTIONS OR COMMENTS ON THE USE OF THE COMPILER, AND PARTICULARLY ANY REPORTS OF BUGS OR SUGGESTIONS FOR IMPROVEMENTS.

SINCE CIMS PL/I IS STILL UNDER DEVELOPMENT, YOU SHOULD NOT GIVE UP ON USING IT IF YOUR INITIAL EXPERIENCE WITH IT IS UNFAVORABLE; WAIT A FEW WEEKS OR MONTHS AND IT WILL BE BETTER. CHECK THE WRITEUP FREQUENTLY FOR CHANGE NOTICES.

THE SUBSET THAT I HAVE IMPLEMENTED IS ORIENTED TOWARDS COMPILER WRITING; IN PARTICULAR, IT IS THAT SUBSET THAT WAS NECESSARY IN ORDER TO WRITE THE COMPILER IN PL/I. SOME ESTIMATE OF THE RELIABILITY OF THE COMPILER MAY BE GLEANED FROM THE FACT THAT IT IS CAPABLE OF COMPILING ITSELF; THE COMPILER PRESENTLY REQUIRES ABOUT 6000 PL/I SOURCE STATEMENTS. THE LACK OF FLOATING POINT INPUT-OUTPUT IS PROBABLY THE MAJOR DIFFICULTY IN USING THE CURRENT VERSION OF THE COMPILER FOR NUMERICAL PROBLEMS, THOUGH FLOATING POINT IS AVAILABLE INTERNALLY FOR COMPUTATION.

THE LANGUAGE ACCEPTED BY CIMS PL/I IS NOT COMPLETELY COMPATIBLE WITH THAT ACCEPTED BY ANY OF THE IBM COMPILERS. DIFFERENCES ARE OF TWO KINDS: CONSTRUCTS THAT ARE IMPLEMENTED IN ONE BUT NOT THE OTHER, AND CONSTRUCTS THAT BEHAVE DIFFERENTLY, OR REQUIRE DIFFERENT SYNTAX, IN THE VARIOUS COMPILERS. THIS LAST SET IS NOT LARGE. INCOMPATIBILITIES ARE DETAILED IN SEC. 5.

2. LANGUAGE FEATURES

IN THIS SECTION I SHALL DESCRIBE THAT PORTION OF PL/I PRESENTLY IMPLEMENTED IN CINS PL/I. A KNOWLEDGE OF PL/I IS NECESSARY IN ORDER TO UNDERSTAND THIS MATERIAL. YOU SHOULD ASSUME THAT ANYTHING NOT MENTIONED IS NOT IMPLEMENTED.

2.1 ATTRIBUTES

2.1.1 SCALAR DATA TYPES

2.1.1.1 FIXED

BASE AND PRECISION ATTRIBUTES ARE ACCEPTED BUT IGNORED. ALL FIXED DATA BEHAVE LIKE FORTRAN INTEGERS.

2.1.1.2 FLCAT

THE PRECISION ATTRIBUTE IS IGNORED. NO I/O FACILITIES EXIST FOR FLOATING NUMBERS, NOR MAY FLOATING CONSTANTS APPEAR IN PROGRAMS. IMPLICIT TYPE CONVERSIONS BETWEEN FIXED AND FLOAT DO EXIST, AND ARE THE ONLY WAY OF OBTAINING OR RETRIEVING FLOAT DATA.

2.1.1.3 CHARACTER

CHARACTER DATA IS FULLY IMPLEMENTED, AND MAY BE VARYING OR NOT. CHARACTER STRING CONSTANTS ARE LIMITED TO 100 CHARACTERS INCLUDING THE APOSTROPHES, BUT VARIABLES ARE NOT LIMITED.

2.1.1.4 BIT

ONLY NONVARYING BIT STRINGS ARE AVAILABLE. THESE MUST HAVE CONSTANT LENGTH (I.E., NOT GIVEN BY AN EXPRESSION). THE LENGTH OF A BIT STRING MUST ALWAYS BE SPECIFIED BY A DECIMAL INTEGER CONSTANT NOT EXCEEDING 60.

2.1.1.5 POINTER

POINTERS MAY ONLY POINT TO WORDS, NOT TO BITS.

2.1.1.6 AREA

ALLOCATIONS MAY BE DONE IN AREAS, BUT AREAS MAY NOT BE ASSIGNED. NOR CAN FREEING BE DONE IN AN AREA. AN AREA MAY BE TOTALLY FREED BY ASSIGNING EMPTY TO IT.

2.1.1.7 LABEL

THE ONLY PERMITTED FORM OF LABEL DATA IS THE CONSTANT LABEL ARRAY. SUCH AN ARRAY IS ALWAYS DECLARED IMPLICITLY THROUGH THE APPEARANCE OF A SUBSCRIPTED

LABEL PREFIX; PRESENTLY, SUCH ARRAYS MUST BE ONE-DIMENSIONAL. IBM PL/I REQUIRE SUCH AN ARRAY TO BE DECLARED IN A DECLARE STATEMENT; CIMS PL/I PROHIBITS SUCH A DECLARATION, CONSTANT LABEL ARRAYS ARE USED TO OBTAIN THE EFFECT OF THE FORTRAN COMPUTED GO TO ,

2.1.1.8 ENTRY

ONLY ENTRY CONSTANTS, EITHER INTERNAL OR EXTERNAL, ARE ALLOWED.
(CF. SEC. 2.1.6 AND 2.6)

2.1.2 AGGREGATION

STRUCTURES AND ARRAYS ARE IMPLEMENTED, AND CAN BE USED IN ASSIGNMENTS. THE REFER OPTION IS RECOGNIZED, WITH THE SAME LIMITATIONS AS F-LEVEL PL/I. CROSS-SECTIONS OF ARRAYS (WITH * SUBSTITUTED FOR A DIMENSION) ARE NOT RECOGNIZED. THE * NOTATION MAY BE USED FOR STRING LENGTHS AND ARRAY BOUNDS OF PARAMETERS.

2.1.3 STORAGE CLASSES

THE AUTOMATIC, STATIC, AND BASED STORAGE CLASSES ARE AVAILABLE. BASED VARIABLES NEED NOT BE DECLARED WITH A POINTER.

2.1.4 SCOPE ATTRIBUTES

THE SCOPE ATTRIBUTES INTERNAL AND EXTERNAL ARE RECOGNIZED. HOWEVER, A GIVEN EXTERNAL VARIABLE CANNOT BE DECLARED MORE THAN ONCE IN A PROCEDURE; AN ATTEMPT TO DO SO WILL CREATE AN ASSEMBLY ERROR. EXTERNAL VARIABLE NAMES AND EXTERNAL ENTRY NAMES LONGER THAN SEVEN CHARACTERS ARE TRUNCATED BY USING THE FIRST FOUR AND LAST THREE CHARACTERS. EXTERNAL VARIABLES ARE IMPLEMENTED AS LABELLED COMMON AREAS.

2.1.5 INITIAL ATTRIBUTE

THE INITIAL ATTRIBUTE IS RECOGNIZED, BUT ITERATION FACTORS MUST BE DECIMAL INTEGER CONSTANTS. FIVE LEVELS OF NESTING ARE ALLOWED, AND THE LENGTH OF A LIST IS ESSENTIALLY UNLIMITED. ITERATED EXPRESSIONS MUST BE CONSTANTS. STRING REPETITION FACTORS ARE NOT ALLOWED.

2.1.6 ENTRY ATTRIBUTE

THE ENTRY ATTRIBUTE MUST BE DECLARED FOR EXTERNAL ENTRY POINTS, AND MUST NOT BE DECLARED FOR INTERNAL ENTRY POINTS. THE ASTERISK NOTATION IS ALLOWED FOR STRING LENGTHS. ENTRY DESCRIPTORS MUST BE OMITTED FOR ANY PARAMETERS THAT ARE AGGREGATES, AND THE ARGUMENTS PASSED TO SUCH PARAMETERS MUST AGREE EXACTLY WITH THE PARAMETER DECLARATION. FOR PARAMETERS, STRING LENGTHS AND ARRAY BOUNDS MUST BE GIVEN EITHER BY DECIMAL CONSTANTS OR BY ASTERISKS (SEE ALSO SEC. 2.6).

2.1.7 FILE ATTRIBUTES

THERE ARE FOUR FILE ATTRIBUTES AVAILABLE: FILE, INPUT, OUTPUT, AND ENVIRONMENT, ENVIRONMENT HAS THE FOLLOWING OPTIONS:

REWIND - NOREWIND - UNLOAD
BINARY - CODED

THE ENVIRONMENT OPTION MAY BE ATTACHED TO A FILE DECLARATION, AN OPEN STATEMENT, OR A CLOSE STATEMENT. AN EXAMPLE OF AN ENVIRONMENT OPTION IS:

ENV(REWIND,CODED)

THE DEFAULT ENVIRONMENTS ARE NOREWIND AND BINARY. IF ANY CONFLICTING ENVIRONMENT OPTIONS ARE PRESENT AT OPEN OR CLOSE TIME, THE UNDEFINED-FILE CONDITION IS RAISED, SEE SEC. 2.8 FOR RELATED DISCUSSION.

2.1.8 BUILTIN ATTRIBUTE

THE BUILTIN ATTRIBUTE MAY BE DECLARED. BUILTIN NAMES THAT ARE DECLARED WITH OTHER ATTRIBUTES MAY BE USED WITHOUT CONFLICT, PROVIDED THAT BUILTIN IS DECLARED FOR ALL BLOCKS CONTAINING USES OF THE BUILTIN FUNCTION IN ITS USUAL SENSE. NOTE THAT NULL MUST BE DECLARED BUILTIN IF IT IS USED FOR THE NULL POINTER, AND SIMILARLY FOR ENTRY.

2.1.9 LIKE ATTRIBUTE

THE LIKE ATTRIBUTE IS AVAILABLE.

2.2 ATTRIBUTE DETERMINATION

CONTEXTUAL DECLARATION IS AVAILABLE FOR FILES, POINTERS, AREAS, AND BUILTIN FUNCTIONS THAT EXPECT ARGUMENTS. UNDECLARED VARIABLES NOT AFFECTED BY CONTEXTUAL DECLARATION ALL BECOME FIXED. THIS BEHAVIOR IS DIFFERENT FROM THAT OF IBM F-LEVEL PL/I, WHICH DEFAULTS VARIABLE TYPES ACCORDING TO THE FIRST LETTER OF THE VARIABLE NAME AS IN FORTRAN.

MISSING ATTRIBUTES ARE FILLED IN BY THE COMPILER USING THE STANDARD PL/I RULES.

2.3 STATEMENT TYPES

THE FOLLOWING STATEMENT TYPES ARE AVAILABLE. WITH THE RESTRICTIONS NOTED:

2.3.1 ASSIGNMENT STATEMENT

ASSIGNMENTS MAY BE MADE TO AGGREGATES AS WELL AS TO SCALARS. MULTIPLE ASSIGNMENT IS PERMITTED. HOWEVER, ALL ITEMS ON THE LEFT SIDE OF AN ASSIGNMENT STATEMENT MUST HAVE IDENTICAL AGGREGATE TYPES, OR AN UNDIAGNOSED ERROR WILL OCCUR. THE BY NAME OPTION IS ACCEPTED BUT IGNORED. IMPLICIT TYPE CONVERSION IS PERFORMED ON ASSIGNMENT WHEN NECESSARY.

2.3.2 ALLOCATE STATEMENT

THE ALLOCATE STATEMENT WITH OPTIONAL IN AND SET CLAUSES IS RECOGNIZED.

2.3.3 BEGIN STATEMENT

THE BEGIN STATEMENT IS ALLOWED.

2.3.4 CALL STATEMENT

THE CALL STATEMENT IS ALLOWED, THOUGH THE TASK, EVENT AND PRIORITY OPTIONS ARE NOT RECOGNIZED.

2.3.5 CLOSE STATEMENT

THE CLOSE STATEMENT IS ALLOWED. LISTS OF FILES ARE ALLOWED, AS IS THE ENVIRONMENT OPTION. AT CLOSE TIME, ANY DECLARED ENVIRONMENT OPTIONS ARE COMBINED WITH THE ENVIRONMENT OPTIONS ATTACHED TO THE CLOSE STATEMENT.

2.3.6 DECLARE STATEMENT

THE DECLARE STATEMENT IS ALLOWED. FACTORING OF ATTRIBUTES, INCLUDING LEVEL NUMBERS AND DIMENSIONS, IS ALLOWED.

2.3.7 DEFAULT STATEMENT

THE DEFAULT STATEMENT IS AVAILABLE. THIS STATEMENT IS DEFINED BY ANSI PL/I, BUT IS NOT IMPLEMENTED IN ANY IBM COMPILERS IN THIS FORM. THE DEFAULT STATEMENT ALLOWS THE USER TO SPECIFY ATTRIBUTES TO BE ASSIGNED TO A VARIABLE, DEPENDING ON THE ATTRIBUTES THAT IT ALREADY HAS AS WELL AS ON THE LEADING CHARACTERS OF THE NAME OF THE VARIABLE. THE DEFAULT STATEMENT IS DESCRIBED IN DETAIL IN APPENDIX A.

2.3.8 DO STATEMENT

ALL OF THE FORMS OF THE DO STATEMENT ARE ALLOWED, BUT THE PARAMETERS OF THE DO ARE ALWAYS CONVERTED TO FIXED VALUES. EXPRESSIONS MAY BE USED AS DO PARAMETERS.

2.3.9 END STATEMENT

THE END STATEMENT IS ALLOWED, AND MULTIPLE END CLOSURE IS RECOGNIZED.

2.3.10 ENTRY STATEMENT

THE ENTRY STATEMENT IS ALLOWED. MULTIPLE ENTRY POINTS ARE PERMITTED, THE RETURNS OPTION IS RECOGNIZED.

2.3.11 FREE STATEMENT

THE FREE STATEMENT IS ALLOWED, BUT VARIABLES ALLOCATED IN AREAS CANNOT BE FREED.

2.3.12 GO TO STATEMENT

THE GO TO STATEMENT IS ALLOWED. IT IS PERMITTED TO GO TO A LABEL IN A BLOCK CONTAINING THE CURRENT ONE. IT IS PERMITTED TO GO TO AN ELEMENT OF A

CONSTANT LABEL ARRAY, USING A SUBSCRIPTED LABEL REFERENCE.

2.3.13 IF STATEMENT

THE IF STATEMENT IS ALLOWED. HOWEVER, ASSIGNMENTS TO THE IDENTIFIER IF ARE NOT PROPERLY RECOGNIZED. IF STATEMENTS MAY HAVE ASSOCIATED ELSE CLAUSES.

2.3.14 NULL STATEMENT

THE NULL STATEMENT IS ALLOWED.

2.3.15 ON STATEMENT

THE ON STATEMENT IS ALLOWED. CONDITIONS PRESENTLY RECOGNIZED ARE:

AREA
ENDFILE
UNDEFINEDFILE
ERROR
FINISH
STORAGE

MULTIPLE CONDITIONS MAY BE SPECIFIED IN A SINGLE ON-STATEMENT, E.G.:

ON ENDFILE(OUT), UNDEFINEDFILE(OUT) GO TO LAB ;

THE BEHAVIOR OF ENDFILE WITH RESPECT TO END-OF-RECORD IS DISCUSSED IN SEC, 2.8. THE STORAGE CONDITION IS RAISED WHEN THE MAIN STORAGE POOL HAS BEEN EXHAUSTED. NORMALLY THERE IS NO WAY TO RECOVER FROM THIS CONDITION. HOWEVER, UNDER SOME CIRCUMSTANCES RECOVERY MAY BE POSSIBLE EITHER BY FREEING LARGE BLOCKS OF ALLOCATED STORAGE OR BY EXITING FROM A NEST OF PROCEDURES USING AUTOMATIC STORAGE.

BECAUSE OF A BUG THAT IS RATHER DIFFICULT TO FIX PRESENTLY, ON-UNITS THAT ARE BEGIN BLOCKS CANNOT DECLARE ANY LOCAL AUTOMATIC VARIABLES. TO BYPASS THIS RESTRICTION, TURN THE BEGIN BLOCK INTO A PARAMETERLESS PROCEDURE AND CALL IT.

2.3.16 OPEN STATEMENT

THE OPEN STATEMENT IS ALLOWED, WITH OPTIONS TITLE, OUTPUT, INPUT, AND ENVIRONMENT (CF. SECS. 2.1.7 AND 2.8). AT OPEN TIME, DECLARED OPTIONS ARE COMBINED WITH THOSE SPECIFIED FOR THE OPEN STATEMENT. LISTS OF FILES ARE ALLOWED.

2.3.17 PROCEDURE STATEMENT

THE PROCEDURE STATEMENT IS ALLOWED, WITH OPTIONS(MAIN) USED TO INDICATE A MAIN PROCEDURE. PROCEDURES NEED NOT HAVE LABELS IF THEY CONTAIN ENTRY STATEMENTS. THE RECURSIVE AND RETURNS OPTIONS ARE RECOGNIZED.

2.3.18 READ STATEMENT

ONLY ONE FORM OF THE READ STATEMENT IS RECOGNIZED: READ FILE(*FILE*) INTO (*VARIABLE*) . A READ STATEMENT ON AN UNOPENED FILE CAUSES THAT FILE TO BE OPENED AS AN INPUT FILE.

2.3.19 RETURN STATEMENT

THE RETURN STATEMENT IS RECOGNIZED; IT HAS AN OPTIONAL ARGUMENT. THIS ARGUMENT MUST BE GIVEN IF THE PROCEDURE WAS ENTERED AT AN ENTRY POINT THAT RETURNS A VALUE, AND MUST BE OMITTED OTHERWISE.

2.3.20 STOP STATEMENT

THE STOP STATEMENT IS RECOGNIZED.

2.3.21 WRITE STATEMENT

ONLY ONE FORM OF THE WRITE STATEMENT IS RECOGNIZED: WRITE FILE(*FILE*) FROM (*VARIABLE*) . A WRITE STATEMENT ON AN UNOPENED FILE CAUSES THAT FILE TO BE OPENED AS AN OUTPUT FILE.

2.4 EXPRESSIONS

EXPRESSIONS ARE IMPLEMENTED, INCLUDING IMPLICIT CONVERSIONS RESULTING FROM THE COMBINATION OF DIFFERENT DATA TYPES THROUGH OPERATORS. VARIABLE REFERENCES MAY BE INCOMPLETELY QUALIFIED IF NO AMBIGUITY RESULTS, AND SUBSCRIPTS IN SUBSCRIPTED QUALIFIED NAMES MAY BE INTERLEAVED IN ANY POSITION PROVIDED THAT THEY ARE GIVEN IN THE RIGHT SEQUENCE. NOTE THAT CROSS-SECTIONS OF ARRAYS ARE NOT ALLOWED.

ANY POINTER-VALUED EXPRESSION MAY BE USED AS A POINTER QUALIFIER; THUS POINTER QUALIFIERS MAY BE NESTED TO ANY DEPTH. THIS GENERALIZATION IS NOT ALLOWED BY F-LEVEL PL/1.

ALL OF THE PL/I OPERATORS ARE ALLOWED. IN PERFORMING COMPARISON OPERATIONS ON CHARACTER STRINGS, THE BLANK CHARACTER IS TREATED AS THOUGH IT HAD DISPLAY CODE 0 (I.E., IT IS THE LOWEST CHARACTER IN THE COLLATING SEQUENCE). OTHERWISE, THE CDC COLLATING SEQUENCE IS USED.

THE ALLOWABLE CONSTANT TYPES ARE SIGNED DECIMAL INTEGERS, BIT STRINGS, AND CHARACTER STRINGS. STRING REPETITION FACTORS ARE NOT RECOGNIZED.

2.5 TYPE CONVERSION

TYPE CONVERSION OCCURS IMPLICITLY IN THREE CONTEXTS:

1. APPLICATION OF OPERATORS

2. ASSIGNMENT

3. MATCHING OF ARGUMENTS TO ENTRY PARAMETERS

THE ONLY FACILITY PRESENTLY AVAILABLE FOR EXPLICIT TYPE CONVERSION IS THE CHAR BUILT-IN FUNCTION, WHICH MAY BE CALLED EITHER WITH OR WITHOUT A SECOND ARGUMENT.

THE FOLLOWING TYPE CONVERSIONS ARE AVAILABLE:

1. FIXED TO FLOAT.
2. FIXED TO CHARACTER. THE RESULT IS AN EIGHT-CHARACTER STRING RIGHT-JUSTIFIED WITH BLANK FILL.
3. FIXED TO BIT. THE RESULT IS A 17-BIT STRING.
4. FLOAT TO FIXED.
5. CHARACTER TO FIXED. THE INPUT CHARACTER STRING MAY HAVE BLANKS ON EITHER SIDE OF THE NUMBER TO BE CONVERTED. ALL BLANKS CONVERT TO ZERO.
6. CHARACTER TO BIT. THE INPUT CHARACTER STRING MUST BE OF CONSTANT LENGTH, AND THAT LENGTH MUST NOT EXCEED 60.
7. BIT TO CHARACTER.
8. BIT TO FIXED.

2.6 FUNCTION CALLS AND PARAMETER PASSAGE

GENERALLY SPEAKING, THE USUAL PL/I RULES APPLY TO PARAMETER PASSAGE AND FUNCTION CALLS. HOWEVER, ENTRY NAMES DECLARED WITH NO RETURNS ATTRIBUTE ALWAYS DEFAULT TO FIXED. A FUNCTION CAN ONLY RETURN A SCALAR AS A VALUE; RETURNED CHARACTER STRINGS MUST BE DECLARED WITH CONSTANT LENGTH. AGGREGATES MAY BE PASSED AS ARGUMENTS, BUT THEY MUST AGREE EXACTLY WITH THE ATTRIBUTES OF THE RECEIVING PARAMETER. NO PROVISION IS MADE FOR THE DECLARATION OF AGGREGATE PARAMETER DESCRIPTORS. IF A PARAMETER IS NOT DECLARED IN AN ENTRY DECLARATION, THEN IT IS ASSUMED THAT THE ARGUMENT MATCHES THE PARAMETER, AND NO CONVERSION IS DONE. IF A PARAMETER IS DECLARED THEN THE ARGUMENT IS CONVERTED TO THE PARAMETER TYPE IF NECESSARY. NOTE THAT FOR INTERNAL PROCEDURES AN ENTRY DECLARATION WITH A FULL SET OF PARAMETER DESCRIPTORS IS ALWAYS CREATED IMPLICITLY.

MULTIPLE ENTRY POINTS ARE ALLOWED, AND THEY NEED NOT AGREE IN THE TYPE OF VALUE RETURNED. THE NECESSARY CONVERSIONS WILL ALWAYS BE PERFORMED. NOTE THAT A PROCEDURE INVOKED BY A FUNCTION REFERENCE MUST RETURN A VALUE, WHILE A PROCEDURE INVOKED BY A CALL STATEMENT MUST NOT RETURN A VALUE. VIOLATION OF THESE RULES CAN LEAD TO MYSTERIOUS RUN-TIME ERRORS.

FUNCTION CALLS ARE PERFORMED USING THE FTN CONVENTIONS. IN PARTICULAR, A1 CONTAINS A POINTER TO THE ARGUMENT LIST AND THE RESULT IS RETURNED IN X6. THUS PL/I PROCEDURES CAN CALL FTN ROUTINES. HOWEVER, PL/I USES DOPE VECTORS TO DESCRIBE CHARACTER STRINGS AND AGGREGATES, SO THESE CAN NEITHER BE PASSED NOR RETURNED TO FTN PROCEDURES WITHOUT A MORE DETAILED KNOWLEDGE OF THE PL/I

RUN-TIME STRUCTURE.

2.7 BUILTIN FUNCTIONS

THE FOLLOWING BUILTIN FUNCTIONS ARE AVAILABLE:

ABS	MIN
CHAR	MOD
EMPTY	NULL
HBOUND	SIGN
INDEX	SUBSTR
LBOUND	UNSPEC
LENGTH	VERIFY
MAX	

THE FOLLOWING RESTRICTIONS SHOULD BE NOTED:

1. HBOUND AND LBOUND CAN ONLY HAVE A CONSTANT AS THEIR SECOND ARGUMENT.
2. EMPTY CAN ONLY BE USED ON THE RIGHT SIDE OF AN ASSIGNMENT TO AN AREA. IF USED, IT MUST BE EXPLICITLY DECLARED TO BE BUILTIN.
3. MOD ALWAYS CONVERTS ITS ARGUMENTS TO FIXED.
4. SUBSTR AND UNSPEC CANNOT BE USED ON THE LEFT SIDE OF AN ASSIGNMENT STATEMENT.
5. IF SUBSTR IS APPLIED TO A BIT STRING, THE THIRD ARGUMENT MUST EITHER BE CONSTANT OR MISSING. IF IT IS MISSING, THE LENGTH OF THE RESULTING BIT STRING IS ALWAYS TAKEN AS 60.
6. UNSPEC CANNOT BE APPLIED TO VARYING STRINGS, SINCE THE RESULT WOULD PRODUCE A VARYING BIT STRING. A DATA TYPE THAT IS NOT YET AVAILABLE, UNSPEC OF A FIXED DATUM YIELDS 17 BITS; UNSPEC OF A POINTER YIELDS 18 BITS; UNSPEC OF A FLOAT DATUM YIELDS 60 BITS.
7. THE SECOND ARGUMENT OF VERIFY MUST BE A SINGLE CHARACTER, AND CANNOT BE ANY OF THE CHARACTERS:

5 2 - 1

2.8 INPUT-OUTPUT

THERE ARE FOUR RELEVANT STATEMENTS FOR INPUT-OUTPUT: OPEN, CLOSE, READ, AND WRITE. STREAM INPUT-OUTPUT IS NOT YET AVAILABLE. FILES CAN BE OPENED IMPLICITLY BY READ OR WRITE STATEMENTS, BUT DECLARED FILE ATTRIBUTES ARE NOT APPLIED WHEN IMPLICIT OPENING OCCURS. IMPLICIT FILE CLOSING TAKES PLACE ON PROGRAM TERMINATION, EITHER NORMAL OR ABNORMAL. HOWEVER, IF THE PROGRAM TERMINATES ON A MACHINE ERROR SUCH AS ADDRESS OUT OF RANGE, FILES WILL NOT BE CLOSED AND, CONSEQUENTLY, MOST OUTPUT WILL BE LOST. NOTE THAT PL/I INPUT-OUTPUT IS NOT CONSISTENT WITH FORTRAN INPUT-OUTPUT BECAUSE THE FETS ARE ACCESSED DIFFERENTLY. THUS, ALTHOUGH PL/I PROGRAMS CAN CALL FORTRAN PROGRAMS, THE FORTRAN PROGRAMS CANNOT DO ANY INPUT-OUTPUT.

DECLARED ENVIRONMENT OPTIONS ARE APPLIED ON BOTH OPENING AND

CLOSING. THUS, IF A FILE IS DECLARED WITH THE ATTRIBUTE ENV(REWIND), IT WILL BE REWOUND ON BOTH OPENING AND CLOSING. NOTE THAT IF NEITHER THE FILE DECLARATION NOR THE OPEN STATEMENT FOR A FILE SPECIFY INPLT OR OUTPUT, THEN INPUT WILL BE ASSUMED. THUS WHEN AN OUTPUT FILE IS OPENED EXPLICITLY, THE OUTPUT ATTRIBUTE MUST BE ATTACHED TO EITHER THE OPEN STATEMENT OR THE FILE DECLARATION.

FILES CAN BE REWOUND WITHIN A PROGRAM BY CLOSING THEM AND THEN REOPENING THEM WITH THE ENVIRONMENT(REWIND) OPTION ON THE OPEN STATEMENT.

SPECIAL CONVENTIONS EXIST, AS IN FORTRAN, CONCERNING EMBEDDED ENDS-OF-RECORD IN INPUT FILES. IF THE ACTUAL INPUT FILE IS NAMED $\neq$ INPUT $\neq$, THEN SUCH END-OF-RECORD MARKS ARE TREATED LIKE END-OF-FILE MARKS. ON ANY OTHER FILE THEY ARE IGNORED. THE DECISION ABOUT THE FILE NAME ARE BASED ON THE NAME THAT EXISTS AFTER MAKING ALL SUBSTITUTIONS DUE EITHER TO THE TITLE OPTION OR TO THE CALL CARD SUBSTITUTION FACILITY.

THE READ AND WRITE STATEMENTS MUST SPECIFY FIXED-LENGTH CHARACTER STRINGS AS THEIR VARIABLES. THESE STRINGS ARE THEN WRITTEN OR READ AS THOUGH THEY WERE IN FORTRAN A-FORMAT, I.E., TRAILING BLANKS ARE ELIMINATED. VARYING CHARACTER STRINGS CANNOT BE USED IN THIS CONTEXT.

ALTHOUGH INPUT-OUTPUT FACILITIES ARE RATHER PRIMITIVE, CERTAIN OTHER FACILITIES OF THE LANGUAGE CAN BE USED TO GREAT ADVANTAGE IN PERFORMING INPUT-OUTPUT. INPUT DATA CAN BE BROKEN DOWN USING THE SUBSTR AND INDEX BUILT-IN FUNCTIONS, WHILE OUTPUT DATA CAN BE BUILT UP USING STRING CONCATENATION. SINCE IMPLICIT NUMBER-TO-CHARACTER AND CHARACTER-TO-NUMBER CONVERSIONS ARE AVAILABLE, IT IS NOT DIFFICULT TO OBTAIN NUMERICAL OUTPUT. A FIXED DATUM, WHEN CONVERTED TO A CHARACTER STRING, PRODUCES AN EIGHT-CHARACTER REPRESENTATION WITH LEADING BLANKS. AS AN EXAMPLE, THE FOLLOWING COULD BE USED TO WRITE OUT THE VALUE OF K TOGETHER WITH IDENTIFYING INFORMATION:

```
DCL C CHAR(80);  
C =  $\neq$  K =  $\neq$ ↑↑K;  
WRITE FILE (OUT) FROM (C);
```

MORE GENERALLY, ONE CAN DEFINE A PROCEDURE PRINT:

```
PRINT: PROC(C);  
    DCL C CHAR(*) ;  
    WRITE FILE(OUT) FROM (C) ;  
END PRINT ;
```

IF PRINT IS PROVIDED AS AN EXTERNAL PROCEDURE, IT MUST BE DECLARED AS:

```
DCL PRINT ENTRY(CHAR(*)) ;
```

PRINT CAN BE CALLED WITH ANY ARGUMENT THAT EITHER IS A CHARACTER-STRING VARIABLE OR CAN BE CONVERTED TO A CHARACTER-STRING VARIABLE, AND THAT ARGUMENT WILL BE PRINTED. NOTICE THAT IF PRINT IS PROVIDED WITH AN ARGUMENT THAT IS NOT A FIXED-LENGTH CHARACTER STRING, THE ARGUMENT WILL IMPLICITLY BE CONVERTED TO A FIXED-LENGTH CHARACTER STRING.

2,9 CHARACTER SET

SINCE THE CDC DISPLAY CHARACTER SET DIFFERS FROM THE STANDARD PL/I CHARACTER SET, A MAPPING IS NECESSARY. THOSE CHARACTERS COMMON TO THE STANDARD PL/I SET AND THE DISPLAY SET ARE UNCHANGED, THREE AREAS OF DIFFERENCE EXIST:

1. THE PL/I APOSTROPHE IS REPRESENTED BY $\neq$.

2. THE NON-ALPHANUMERIC CHARACTERS ALLOWED IN IDENTIFIERS ARE:

\$ * %

3. THE OPERATORS THAT DIFFER FROM STANDARD PL/I ARE:

+ OR
++ CONCATENATE
^ AND

2.10 KEY WORDS AND PHRASES

THE FOLLOWING KEYWORDS AND KEY PHRASES ARE RECOGNIZED, WITH THE INDICATED ABBREVIATIONS:

ALLOCATE	
AUTOMATIC	AUTO
AREA	
BASED	
BEGIN	
BIT	
BINARY	BIN
BUILTIN	
BY NAME	
BY	
CALL	
CHARACTER	CHAR
CLOSE	
DECIMAL	DEC
DECLARE	DCL
DEFAULT	DFT
DO	
ELSE	
END	
ENDFILE	
FINISH	
ENTRY	
ENVIRONMENT	ENV
ERROR	
EXTERNAL	EXT
FILE	
FIXED	
FLOAT	
FREE	
FROM	
GO TO	GOTO
IF	
INITIAL	INIT
IN	
INTERNAL	INT
INTO	
ON	

OPEN
OPTIONS
PROCEDURE PROC
POINTER PTR
READ
RECURSIVE
RETURNS
SET
STATIC
STOP
STORAGE
TO
UNDEFINEDFILE UNDEF
VARYING VAR
WHILE
WRITE

3. OPERATING ENVIRONMENT

COMPILATION OF A PL/I PROGRAM INVOLVES TWO STEPS: THE COMPILATION PROPER AND THE ASSEMBLY OF THE RESULTING COMPASS CODE. COMPILATION AND EXECUTION TOGETHER REQUIRE THREE SEPARATE DATASETS: THE COMPILER ITSELF, THE SYSTEM TEXT FOR THE ASSEMBLY, AND THE RUN-TIME LIBRARY. TWO OF THESE DATASETS ARE AVAILABLE ON SYSLIB:

PL1 THE COMPILER ITSELF

PL1TEXT THE SYSTEM TEXT REQUIRED FOR ASSEMBLY

THE THIRD DATASET IS A PERMANENT FILE:

PL1RLIB THE RUN-TIME LIBRARY

THE COMPILER PRESENTLY REQUIRES AN ABSOLUTE MINIMUM OF 101K TO LOAD AND EXECUTE. A SIZE OF 110K IS RECOMMENDED. IF STORAGE IS INADEQUATE, THE MESSAGE *MEM REQUEST EXCEEDS JOBCARD FL* WILL APPEAR. THE LARGER THE PROCEDURE TO BE COMPILED, THE MORE SPACE IT WILL REQUIRE. COMPILATION SPEED CAN BE ESTIMATED AT APPROXIMATELY 15 STATEMENTS/SECOND, NOT INCLUDING ASSEMBLY TIME. ASSEMBLY TIME IS APPROXIMATELY 30 PERCENT LONGER THAN COMPILATION TIME, SO THE COMBINED RATE IS APPROXIMATELY 6 STATEMENTS/SECOND.

THE SOURCE LISTING INCLUDES A MESSAGE GIVING THE AMOUNT OF FIELD LENGTH REQUIRED FOR COMPILATION. TO DETERMINE THE CORRECT FIELD LENGTH, USE A GENEROUS INITIAL ESTIMATE AND THEN CUT DOWN ACCORDINGLY. THE COMPILER WILL NOT ATTEMPT TO INCREASE ITS FIELD LENGTH BEYOND THAT EXISTING AT THE TIME IT WAS FIRST CALLED. HOWEVER, EACH EXTERNAL PROCEDURE IS COMPILED IN THE MINIMUM NECESSARY FIELD LENGTH.

THE COMPILATION PROCESS USES FIVE LOCAL FILES:

INPUT SOURCE PROGRAM INPUT

OUTPUT SOURCE PROGRAM LISTING WITH STATEMENT NUMBERS. FOLLOWED BY ERROR LISTING

CODE GENERATED ASSEMBLY CODE

LGO LOADABLE FILE PRODUCED BY THE ASSEMBLER

PL1ERR TEMPORARY FILE USED FOR LIST OF SOURCE PROGRAM ERRORS

3.1 INPUT FORMAT

INPUT TO THE COMPILER IS ASSUMED TO BE IN COLUMNS 2 THROUGH 72. THE PRINTED LISTING ACTUALLY GOES UP TO COLUMN 90, SO INPUT OBTAINED FROM UPDATE WILL SHOW THE UPDATE SERIALIZATION. CHARACTERS APPEARING IN COLUMN 1 ARE USED FOR CARRIAGE CONTROL IN THE LISTING.

EACH EXTERNAL PROCEDURE MUST BE FOLLOWED BY A CARD WITH A * IN COLUMN 1 TO INDICATE THE END OF THE PROCEDURE. THIS CONVENTION IS NEEDED SINCE COMMENTS MAY PRECEDE THE INITIAL PROCEDURE STATEMENT OR FOLLOW THE FINAL END STATEMENT.

3.2 CONTROL CARD SETUP

AS AN EXAMPLE, CONSIDER A PL/I PROGRAM WITH THE FOLLOWING CHARACTERISTICS:

- 1. THE PROGRAM CONSISTS OF TWO EXTERNAL PROCEDURES.
- 2. THE PROGRAM READS ITS INPUT FROM FILE IN AND PRODUCES ITS OUTPUT ON FILE OUT.
- 3. THE PROGRAM DOES NOT REQUIRE MORE THAN 50K TO EXECUTE.

THE FOLLOWING CONTROL CARD SEQUENCE IS THEN APPROPRIATE:

```
A123456,CM110000,T100. YOUR NAME
LOADER,
NOREDUCCE,
CIMGET(PL1,PL1TEXT)
ATTACH,RLIB,PL1RLIB.
PL1.
RFL,50000,
REWIND(CODE)
COMPASS(I=CODE,G=PL1TEXT,L=0,O=0)
LOAD,LGO,RLIB.
EXECUTE,,IN=INPUT,OUT=OUTPUT.  #OPTIONAL DATA FOR MAIN PROCEDURE#
EOR
-----
-----FIRST PL/I PROCEDURE-----
-----
*
-----
-----SECOND PL/I PROCEDURE-----
-----
*
EOR
-----
-----INPUT DATA-----
-----
EOF
```

NOTE THE FOLLOWING REMARKS ON THIS SEQUENCE:

- 1. THE OUTPUT WILL APPEAR IN THE SEQUENCE:
 - , LISTING OF SOURCE PROGRAM, WITH ERRORS
 - , RESULTS OF EXECUTING THE PROGRAM
- 2. EACH PROCEDURE MUST BE FOLLOWED BY A STAR IN COLUMN 1.
- 3. TO CALL THE COMPILER WITH AN ALTERNATE OUTPUT FILE OF -OUT1- AND AN ALTERNATE INPUT FILE OF -IN1-, USE THE CALL CARD
PL1(INPUT=IN1,OUTPUT=OUT1)

3.3 COMPILER OPTIONS

COMPILER OPTIONS ARE SPECIFIED AS AN ARGUMENT TO THE COMPILER, USING THE CONVENTION OF SEC. 3.4.2. FOR INSTANCE, THE CALLING CARD

PL1. *XREF*

TURNS ON THE -XREF- OPTION. OPTIONS ARE SEPARATED BY COMMAS IN THE OPTIONS LIST,

3.3.1 XREF

THE -XREF- OPTION CAUSES A CROSS-REFERENCE MAP AND ATTRIBUTE LISTING TO APPEAR IN THE SOURCE LISTING. NOTE THAT EXTRA COMPILATION SPACE IS REQUIRED WHEN THIS OPTION IS USED.

3.3.2 ST,NST

WHEN THE -ST- OPTION IS USED, REGISTER A0 IS SET TO THE CURRENT STATEMENT NUMBER AT THE START OF EXECUTING EACH NON-NULL STATEMENT. THE DEFAULT IS -ST-, WHICH TURNS THE OPTION ON. NOTE THAT THIS OPTION CAUSES THE OBJECT PROGRAM TO RUN SLOWER AND TAKE MORE SPACE. IT IS USEFUL, HOWEVER, FOR DEBUGGING. -NST- TURNS THE OPTION OFF.

3.4 PROGRAM CALLING CARD

THE CARD THAT CALLS A COMPILED PL/I PROGRAM CAN ALSO BE USED TO ACHIEVE SUBSTITUTION OF FILE NAMES AND PASSAGE OF AN ARGUMENT TO THE CALLED PROGRAM. THESE FACILITIES ARE DESCRIBED IN THE FOLLOWING TWO SECTIONS.

3.4.1 FILE NAME SUBSTITUTION

THE FILES REFERENCED INSIDE A PROGRAM ARE VIRTUAL FILES. AN ACTUAL NAME CAN BE SUBSTITUTED FOR A VIRTUAL NAME AT THE TIME THAT A PL/I PROGRAM IS CALLED. THE PROGRAM CALL:

```
LOAD(LGO,RLIB)
EXECUTE,,A1=B1,A2=B2,A3=B3,
```

CAUSES THE ACTUAL FILE NAMES B1,B2, B3 TO BE SUBSTITUTED FOR THE VIRTUAL FILE NAMES A1,A2, AND A3 RESPECTIVELY. IF THE PROGRAM REFERENCES FILE A1 INTERNALLY FOR OUTPUT, THEN THE ACTUAL OUTPUT WILL APPEAR ON FILE B1. IF NO SUBSTITUTION IS GIVEN FOR A VIRTUAL FILE NAME, THE ACTUAL NAME IS TAKEN TO BE THE SAME AS THE VIRTUAL NAME. NOTE THAT FILE NAME SUBSTITUTION MAY BE USED FOR CALLS TO THE COMPILER ITSELF IN THE SAME FASHION. NOTE ALSO THAT IF A PROGRAM IS INVOKED BY AN EXECUTE CARD, AN EXTRA COMMA IS NEEDED FOLLOWING EXECUTE.

IF AN ACTUAL FILE NAME IS -0-, THEN THAT FILE IS TREATED AS A NULL FILE. ANY OUTPUT SENT TO THAT FILE IS DISCARDED; ANY ATTEMPT TO READ THE FILE WILL RAISE THE ENDFILE CONDITION.

3.4.2 PASSING AN ARGUMENT TO A MAIN PROCEDURE

AN ARGUMENT MAY BE PASSED TO THE MAIN PROCEDURE FROM THE PROGRAM CALL CARD. TO ACCOMPLISH THIS, DECLARE THE MAIN PROCEDURE TO HAVE A SINGLE PARAMETER. THAT PARAMETER MUST THEN BE DECLARED TO BE CHARACTER(80) VARYING. ON THE PROGRAM CALL OR EXECUTE CARD, THE ARGUMENT TO BE PASSED TO THE MAIN PROCEDURE IS SURROUNDED BY THE CHARACTER *, AND WRITTEN FOLLOWING THE CLOSING

PARENTHESIS OR PERIOD OF THE CALL. EMBEDDED * CHARACTERS WITHIN THE ARGUMENT ARE WRITTEN AS ** . IF NO ARGUMENT IS PROVIDED, A NULL STRING WILL BE PASSED AS ARGUMENT. FOR EXAMPLE, THE FOLLOWING PROGRAM CALL CARD USES BOTH FILE NAME SUBSTITUTION AND PARAMETER PASSAGE:

```
LOAD,LGO,RLIB,  
EXECUTE,,INPUT=TAPE1, *THIS WON**T FAIL*  
LGO(INPUT=TAPE1) *THIS WON**T FAIL*
```

4. ERROR MESSAGES AND DEBUGGING TECHNIQUES

IF YOUR PROGRAM DOES NOT SUCCESSFULLY COMPILE AND EXECUTE, THE FIRST THING YOU MUST DO IS TO TRY TO CLASSIFY THE CAUSE. THE FOLLOWING KINDS OF ERRORS CAN OCCUR:

1. SOURCE PROGRAM ERRORS CORRECTLY DIAGNOSED BY THE COMPILER,
2. SOURCE PROGRAM ERRORS INCORRECTLY DIAGNOSED BY THE COMPILER,
3. FAILURE OF THE COMPILER TO COMPLETE EXECUTION,
4. ERRORS IN THE COMPASS ASSEMBLY,
5. INCORRECT RESULTS OF EXECUTION DUE TO ERRORS IN THE SOURCE PROGRAM,
6. INCORRECT RESULTS OF EXECUTION DUE TO ERRORS IN THE COMPILER OR THE RUN-TIME LIBRARY.

IF THE COMPILER DIAGNOSES AN ERROR, THEN YOU SHOULD CHECK WHETHER YOU HAVE VIOLATED SOME LANGUAGE RESTRICTION. IF NOT, THEN YOU HAVE A BUG TO REPORT.

ASSUMING THAT THE COMPILER HAS NOT DIAGNOSED ANY ERRORS, THE COMPILER MAY FAIL DURING EXECUTION. IF THIS SITUATION ARISES, THE FIRST CAUSE TO CHECK IS YOUR DECK SETUP. IF THE DECK SETUP IS CORRECT, THEN REPORT A BUG. THE SAME REMARK APPLIES TO ERRORS IN THE COMPASS ASSEMBLY. NOTE THAT WARNING MESSAGES IN THE COMPASS ASSEMBLY ARE EXPECTED. THERE IS ONE LANGUAGE RESTRICTION WHOSE VIOLATIONS WILL LEAD TO ASSEMBLY ERRORS: DECLARATION OF THE SAME EXTERNAL VARIABLE IN MORE THAN ONE PLACE IN A SINGLE EXTERNAL PROCEDURE.

ASSUME THEN THAT YOUR PROGRAM HAS COMPILED AND ASSEMBLED WITHOUT APPARENT DIFFICULTY. IT MAY FAIL DURING EXECUTION EITHER BY FAILING TO COMPLETE EXECUTION OR BY GIVING WRONG ANSWERS. BUGS OCCURRING AT THIS STAGE MAY WELL BE SOURCE PROGRAM ERRORS WITHOUT THAT BEING OBVIOUS. IN ORDER TO ISOLATE THEM, IT IS RECOMMENDED THAT YOU WRITE OUT SUFFICIENTLY MANY INTERMEDIATE RESULTS TO ENABLE YOU TO ISOLATE WHERE THE TROUBLE IS OCCURRING. IF YOU CAN ISOLATE THE TROUBLE AS OCCURRING IN A SINGLE STATEMENT, THEN REPORT THE BUG.

PRESENTLY THE ONLY RUN-TIME MESSAGES ARE THOSE ASSOCIATED WITH PROGRAM ABORTS. AMONG THE CAUSES OF THESE ARE: EXHAUSTION OF THE STORAGE POOL, CONVERSION ERRORS IN GOING FROM CHARACTER STRINGS TO OTHER DATA TYPES, AND READING PAST AN END OF FILE.

IF THE STATEMENT OPTION IS NOT TURNED OFF (CF. SEC. 3.3.2), THEN IT IS POSSIBLE TO DETERMINE THE STATEMENT NUMBER OF THE STATEMENT BEING EXECUTED AT THE TIME OF AN ERROR. WITH THIS OPTION ON, REGISTER AN IS SET TO THE CURRENT STATEMENT NUMBER AT THE BEGINNING OF EXECUTING EACH STATEMENT. THE DUMP SHOULD THEN INDICATE THE STATEMENT NUMBER THAT CAUSED THE DIFFICULTY. HOWEVER, THIS OPTION CAUSES THE PROGRAM TO RUN LONGER AND CONSUME EXTRA SPACE, SO IT SHOULD NOT BE USED FOR DEBUGGED PROGRAMS.

5. AREAS OF INCOMPATIBILITY

THE AREAS OF INCOMPATIBILITY BETWEEN CIMS PL/I AND IBM PL/I ARE ALL MENTIONED IN THE LANGUAGE SPECIFICATION OF SECTION 2. SINCE THE LANGUAGE FEATURES OF CIMS PL/I ARE PRESENTLY FAR FEWER THAN THOSE IN IBM PL/I, THOSE FEATURES THAT ARE LACKING WILL NOT BE DETAILED HERE. HOWEVER, DIFFERENCES IN THE OTHER DIRECTION SHOULD BE NOTED.

5.1 EXTENSIONS IN CIMS PL/I

1. A PROCEDURE STATEMENT NEED NOT BE LABELLED AS LONG AS AT LEAST ONE ENTRY POINT OF THE PROCEDURE IS LABELLED.
2. A POINTER-QUALIFYING EXPRESSION IS NOT RESTRICTED TO BE A SCALAR UNSUBSCRIPTED VARIABLE; IT MAY BE AN ARBITRARY POINTER-VALUED EXPRESSION.
3. THE BASED ATTRIBUTE NEED NOT HAVE AN ASSOCIATED POINTER EXPRESSION. IF SUCH AN EXPRESSION IS GIVEN, IT IS NOT RESTRICTED TO BE A SIMPLE VARIABLE.
4. BASED CHARACTER STRINGS MAY BE VARYING. F-LEVEL PL/I DOES NOT ALLOW THIS.
5. THE ON-STATEMENT MAY SPECIFY SEVERAL CONDITIONS. F-LEVEL PL/I DOES NOT ALLOW THIS.
6. CIMS PL/I HAS A DEFAULT STATEMENT THAT DOES NOT EXIST AT ALL IN THE F-COMPILER. THE IBM OPTIMIZING COMPILER DOES HAVE A DEFAULT STATEMENT, BUT IT HAS DIFFERENT SPECIFICATIONS THAN THE ONE IN CIMS PL/I.
7. CIMS PL/I HAS A STORAGE ON-CONDITION THAT HAS NO EQUIVALENT IN THE F-COMPILER. THE IBM OPTIMIZING COMPILER DOES HAVE SUCH A CONDITION, HOWEVER.

5.2 INCOMPATIBILITIES BETWEEN CIMS PL/I AND IBM PL/I

1. IN CIMS PL/I, INTERNAL PROCEDURES MUST NOT BE DECLARED. IN SOME VERSIONS OF IBM PL/I, THEY MUST BE DECLARED.
2. IN CIMS PL/I, CONSTANT LABEL ARRAYS MUST NOT BE DECLARED; IN IBM PL/I THEY MUST BE DECLARED.
3. IN CIMS PL/I, ALL UNDECLARED VARIABLES WITHOUT CONTEXTUAL ATTRIBUTES DEFAULT TO FIXED (BINARY). IN IBM PL/I THEY WOULD DEFAULT TO FLOAT.
4. THERE IS NO CONNECTION BETWEEN THE ENVIRONMENT OPTION AS USED IN IBM PL/I AND THE ENVIRONMENT OPTION AS USED IN CIMS PL/I.

6. FUTURE PLANS

EVENUALLY I HOPE TO IMPLEMENT THE FULL PL/I LANGUAGE AS DEFINED BY COMMITTEE X3J1 OF THE AMERICAN NATIONAL STANDARDS INSTITUTE. THE LANGUAGE AS PRESENTLY IMPLEMENTED IS A LONG WAY FROM THAT GOAL. THE FEATURES MENTIONED IN REVISION B OF THIS MANUAL HAVE ALL BEEN IMPLEMENTED. STREAM INPUT-OUTPUT IS ON THE WAY, BUT NO OTHER IMPROVEMENTS ARE PLANNED FOR THE IMMEDIATE FUTURE.

7. ACKNOWLEDGMENTS

LINDA SUSKIND CONSTRUCTED THE FACILITIES FOR SUBSTITUTION OF FILE NAMES.

ABRAHAM GETZLER CONSTRUCTED THE CROSS-REFERENCE MAP AND ATTRIBUTE LISTING, AS WELL AS THE OVERLAY ROUTINES USED WITHIN THE COMPILER.

IRA KAYE BUILT THE ENTIRE MECHANISM FOR THE DEFAULT STATEMENT, AND ALSO WROTE APPENDIX A OF THIS MANUAL.

EMILY EISENBERG CONSTRUCTED THE ORIGINAL VERSION OF THE LEXICAL SCANNER.

PETER MACLEAN PROVIDED AN INTERACTIVE DEBUGGING FACILITY WHOSE USE WAS INVALUABLE IN DEBUGGING THE COMPILER.

APPENDIX A

A.1 DESCRIPTION

A.1.1 DEFAULT/DFT (PREDICATE) DFT-ATTR-SET [,DFT-ATTR-SET] ;

IF A VARIABLE MEETS THE CONDITIONS SPECIFIED BY THE PREDICATE, GIVE THE VARIABLE THE ATTRIBUTES SPECIFIED IN THE DFT-ATTR-SETS. ATTEMPT TO APPLY THE SETS IN THE ORDER LISTED, AND APPLY EACH ONE WHICH DOES NOT CAUSE ANY CONFLICTING ATTRIBUTES TO BE GIVEN.

A.1.2 DEFAULT/DFT (PREDICATE) ERROR;

IF THE VARIABLE MEETS THE CONDITIONS SPECIFIED IN PREDICATE, GIVE IT THE ERROR ATTRIBUTE, AND PRINT A MESSAGE. PROGRAM WILL NOT BE EXECUTABLE IF ANY VARIABLES ARE GIVEN THE ERROR ATTRIBUTE.

A.1.3 DEFAULT/DFT SYSTEM;

APPLY THE STANDARD PL/1 SYSTEM DEFAULTS TO THE VARIABLE. ANY DEFAULT STATEMENTS APPEARING AFTER THIS TYPE WILL BE IGNORED.

A.1.4 DEFAULT/DFT NONE ;

APPLY NO FURTHER DEFAULTS, INCLUDING THE SYSTEM DEFAULT. IF A VARIABLE IS NOT COMPLETE, IT WILL BE GIVEN THE ERROR ATTRIBUTE, AND AN ERROR MESSAGE WILL BE PRINTED.

A.2 ORDER OF PROCESSING AND SCOPING.

DEFAULT STATEMENTS ARE PROCESSED IN THE ORDER IN WHICH THEY APPEAR. THE DEFAULT STATEMENTS APPEARING IN A BLOCK ARE APPLIED IN SEQUENCE TO EACH VARIABLE DECLARED IN THE BLOCK.

DEFAULT STATEMENTS FOLLOW THE SAME NAME SCOPING CONVENTIONS AS VARIABLES. THEY ARE KNOWN IN THE BLOCKS IN WHICH THEY APPEAR AND ALL ENCLOSED BLOCKS. THEY ARE PROCESSED FROM THE INNERMOST BLOCK OUT.

EX, 1.

```
A: PROC OPTIONS (MAIN) ;  
  DFT (P1) SET1 ;  
  DCL A1 ;  
  
  ...  
  B: PROC ;  
    DFT (P2) SET2 ;  
    DCL B1 ;  
  
    ...  
  END B ;  
END A ;
```

B1 WILL HAVE PREDICATES P2 AND P1 APPLIED TO IT IN THAT ORDER. A1 WILL HAVE PREDICATE P1 APPLIED TO IT. IF IT IS DESIRED THAT ONLY PREDICATE P2 BE APPLIED TO B1, THEN DFT SYSTEM; SHOULD BE PLACED AFTER DFT(P2) SET2;

EX, 2.

```
A: PROC OPTIONS(MAIN) ;  
  DFT (P3) SET3 ;  
  DFT NONE;
```

```

      ...
B: PROC ;
  DFT SYSTEM ;
  ...
      C : PROC ;
        DFT(P4) SET4 ;
      ...
      END C;
    END B;
  END A;

```

VARIABLES DECLARED IN BLOCK A WILL HAVE PREDICATE P3 APPLIED, AND NO OTHER DEFAULTS, INCLUDING SYSTEM DEFAULTS.
 VARIABLES DECLARED IN BLOCK B WILL HAVE THE STANDARD SYSTEM DEFAULTS APPLIED.
 VARIABLES DECLARED IN BLOCK C WILL HAVE P4 APPLIED TO THEM, AND THEN SYSTEM DEFAULTS.

A.3 DESCRIPTION OF PREDICATES AND DEFAULT ATTRIBUTE SETS.

A.3.1 PREDICATES

PREDICATES ARE MADE UP BY COMBINING ANY OF THE ALLOWABLE ATTRIBUTE KEYWORDS IN BOOLEAN EXPRESSIONS USING ^, !, AND ~. TO REFER TO DIMENSIONED ARRAYS, USE THE KEYWORD DIMENSION OR DIM. IN ADDITION THE FOLLOWING ELEMENTS ARE ALLOWED IN PREDICATES:

RANGE(STRING) - A VARIABLE MEETS THIS CONDITION IF IT STARTS WITH THE CHARACTER STRING.

RANGE(CHAR1 : CHAR2) - A VARIABLE MEETS THIS CONDITIONS IF ITS FIRST LETTER IS IN THE LEXICAL INTERVAL SPECIFIED BY CHAR1 AND CHAR2.

RANGE(*) - ALL VARIABLES MEET THIS CONDITION.

A.3.2 DEFAULT ATTRIBUTE SETS.

ATTRIBUTE SETS ARE MADE UP OF LISTS OF ATTRIBUTES WHICH ARE TO BE APPLIED TO A VARIABLE IF IT MEETS THE CONDITIONS OF THE PREDICATE. AS MANY SETS AS DESIRED MAY BE PLACED IN A DEFAULT STATEMENT. THE ATTRIBUTES GIVEN IN THE SET ARE APPLIED TO THE VARIABLE, PROVIDED THAT THEY DO NOT CONFLICT WITH ATTRIBUTES THAT HAVE ALREADY BEEN ASSIGNED TO THE VARIABLE. IF SUCH CONFLICTS EXIST, THEN THE SET IS IGNORED. ANY ATTRIBUTE OTHER THAN THE LIKE ATTRIBUTE CAN APPEAR IN A DEFAULT ATTRIBUTE SET.

Appendix B. Useful Coding Tricks

In this appendix I shall describe some coding tricks that I used in the compiler and in its run-time library. This section assumes a knowledge of 6600 machine language. In the following descriptions, the assumption is made that any register not defined by the code specification can be used as a temporary.

B.1 Absolute Value, Maximum, and Minimum

The method for taking absolute values without a branch instruction has been passed down by oral tradition, but I have never seen it in print. The following code takes the absolute value of the contents of X7 and leaves the result in X7.

BX6	X7	
AX6	60	FILL X6 WITH SIGN BIT OF X7
BX7	X7-X6	INVERT X7 IF NEGATIVE

A variation of this device will take the maximum of X6 and X7, leaving the result in X7:

FX5	X7-X6	
BX4	X5	
AX4	60	
BX5	X5*X4	X7-X6 IF NEGATIVE, ELSE 0
FX7	X7-X5	IF X7 EXCEEDS X6, SUBTRACT X7-X6

If 60 happens to be available in a B-register, these sequences can each be shortened by one instruction. The code for the minimum is analogous.

B.2 Least Integer in X

If we have a floating number and wish to find the least integer not exceeding that number, the code is tricky because float-to-integer conversion truncates towards zero and gives the wrong result for negative numbers. Assuming input in X7, the following code puts the desired least integer into X7.

MX1	1	
BX2	X7*X1	
LX2	1	1 TO X2 IF X7 NEGATIVE, ELSE 0
IX7	X7+X2	IF X7 NONNEGATIVE, LEAVE IT UNCHANGED
		IF X7 AN EXACT INTEGER, BUMP IT UP
		IF X7 NOT EXACT INTEGER, KEEP SAME
		INTEGER VALUE
UX7	X7,B2	
LX7	X7,B2	CONVERT FLOAT TO INTEGER
IX7	X7-X2	SUBTRACT 1 IF NEGATIVE

To understand this code, note that if X7 is nonnegative the addition and subtraction of X2 has no effect, since X2 contains zero. Otherwise we have two cases. If X7 contained the floating representation of a negative integer, then the addition will give us a value that will, when truncated, give the next higher integer. (E.g., we will go from -3 to -2.99999999.) After conversion to float we have a number one too big (e.g., -2) and the following subtraction corrects this situation. Suppose, on the contrary, that X7 contained the floating representation of a negative number that was not an exact integer. Then we would go, for instance, from -3.3 to -3.299999999 to -3 (after conversion to integer) to -4 (after subtraction).

The same device works for finding the greatest integer. In order to treat zero correctly the correct procedure is to complement X7, use the code given above, and recompute X7.

B.3 Division by Ten

Performing integer division by 10 is necessary when finding the number of words needed to represent a packed character string, since each character occupies 6 bits and each word has 60 bits. Integer division on the 6600 can only be carried out by using the floating division instruction; the whole process requires five instructions. The division instruction itself takes 20 minor cycles, twice as long as any other non-branching instruction. Thus it is desirable to avoid use of the division instruction in dividing by 10. This can be done by multiplying by a suitably unnormalized one-tenth. Assuming input and result in X7, we have:

SA1	-17170631463146314632B	1/10+EPSILON
PX6	X7	FLOAT X7 INTO X6
FX6	X1*X6	QUOTIENT TO X6
SX7	X6	CLEAR JUNK BITS FROM EXPONENT

The exponent is chosen so that the desired quotient appears in the right end of the upper half of the product. A similar trick can be used for division by other integer constants.

B.4 Changing the Representation of Blank

The internal character code for blank is 55B, which gives the wrong result in character comparisons since it places blank after the letters or digits. Before doing a string comparison other than equality or inequality it is desirable to map all blanks to the code 00B, thus redefining the collating sequence to make it more reasonable. It is possible to perform this operation on an entire word without looping. The process proceeds in five stages. In the first stage we map the input word so that 55B characters become 77B (all ones) and no other characters become 77B. In the second stage we convert each 77B to a character such that the leftmost bit is 1, and each other character to a character such that the leftmost bit is 0. This is done through shifting and "and"-ing. In the third stage we convert each 77B to 40B and each other character to 0. In the fourth stage we convert each 40B to 77B and each other character to 0. In the fifth stage we use the mask we have just developed to change the 55B's to 00B's. Assuming input and output in X7, the code is:

SA1	=10H	
BX6	-X1-X7	CHANGE 55B to 77B in X6
BX5	X6	
LX5	3	
BX6	X5*X6	AND 3 LEFT BITS WITH 3 RIGHT BITS
SB1	1	
LX5	X6,B1	
BX6	X5*X6	
LX5	X6,B1	
BX6	X5*X6	
LX5	X6,B1	
BX6	X5*X6	ALL 6 BITS NOW ADDED INTO LEFT BIT
SA2	=40404040404040404040B	
BX6	X2*X6	ELIMINATE EXTRA BITS
AX5	X6,B1	CHANGE 40B TO 77B
BX6	X5+X6	
AX5	X6,B1	
BX6	X5+X6	
AX5	X6,B1	
BX6	X5+X6	
BX5	X6	
AX5	3	
BX6	X5+X6	NOW ALL CHARS ARE 77B or 0
BX7	-X6*X7	MASK 55B TO 0

The same approach can be used, for instance, to count null characters in a word; in this case one produces 40B for each null character and then uses the population count instruction. Other related tasks can also be accomplished with this approach.

B.5 Searching for a Set of Characters in a String

The trick for searching for a set of characters in a string is needed in connection with the VERIFY built-in function in PL/I. In this situation we are given a set of characters, and we wish to find the first character in the string that is not in the set. (The problem of finding the first character that is in the set is similar.) Rather than give the actual code, I will just describe the method. First, form a bit mask such that bit *i* is one if and only if the character with representation *i* is in the set. Then loop through the characters in the string. For each character, use the character representation as a shift count and shift a single one bit left by that count. Form the "and" of the shifted bit and the bit mask constructed above. If the result is nonzero, then the character is in the set.

There is a difficulty with this method. If the characters include any whose octal representations exceed 60, then there will be an ambiguity, and two bit masks are needed rather than one.

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

- A. Makes any warranty or representation, express or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or
- B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

